

# ***Lingo Dictionary***

---

***Macromedia Director MX***



## Trademarks

Afterburner, AppletAce, Attain, Attain Enterprise Learning System, Attain Essentials, Attain Objects for Dreamweaver, Authorware, Authorware Attain, Authorware Interactive Studio, Authorware Star, Authorware Synergy, Backstage, Backstage Designer, Backstage Desktop Studio, Backstage Enterprise Studio, Backstage Internet Studio, Contribute, Design in Motion, Director, Director Multimedia Studio, Doc Around the Clock, Dreamweaver, Dreamweaver Attain, Drumbeat, Drumbeat 2000, Extreme 3D, Fireworks, Flash, Fontographer, FreeHand, FreeHand Graphics Studio, Generator, Generator Developer's Studio, Generator Dynamic Graphics Server, Knowledge Objects, Knowledge Stream, Knowledge Track, Lingo, Live Effects, Macromedia, Macromedia M Logo & Design, Macromedia Contribute, Macromedia Flash, Macromedia Xres, Macromind, Macromind Action, MAGIC, Mediamaker, Object Authoring, Power Applets, Priority Access, Roundtrip HTML, Scriptlets, SoundEdit, ShockRave, Shockmachine, Shockwave, Shockwave Remote, Shockwave Internet Studio, Showcase, Tools to Power Your Ideas, Universal Media, Virtuoso, Web Design 101, Whirlwind and Xtra are trademarks of Macromedia, Inc. and may be registered in the United States or in other jurisdictions including internationally. Other product names, logos, designs, titles, words or phrases mentioned within this publication may be trademarks, servicemarks, or tradenames of Macromedia, Inc. or other entities and may be registered in certain jurisdictions including internationally.

This guide contains links to third-party Web sites that are not under the control of Macromedia, and Macromedia is not responsible for the content on any linked site. If you access a third-party Web site mentioned in this guide, then you do so at your own risk. Macromedia provides these links only as a convenience, and the inclusion of the link does not imply that Macromedia endorses or accepts any responsibility for the content on those third-party sites.

## Apple Disclaimer

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

Copyright © 2002 Macromedia, Inc. All rights reserved. This manual may not be copied, photocopied, reproduced, translated, or converted to any electronic or machine-readable form in whole or in part without prior written approval of Macromedia, Inc. Third Party Software Notices and/or Additional Terms and Conditions can be found at <http://www.macromedia.com/go/thirdparty/>. Part Number ZDR90M200

## Acknowledgments

Writing: Jay Armstrong, George Brown, Stephanie Gowin, and, Tim Statler

Editing: Rosana Francescato, Mary Ferguson, Mary Kraemer, and Noreen Maher

Project Management: Stuart Manning

Production: Chris Basmajian, Caroline Branch, John Francis, and Patrice O'Neill

Multimedia: Aaron Begley and Benjamin Salles

First Edition: December 2002

Macromedia, Inc.  
600 Townsend St.  
San Francisco, CA 94103

# CHAPTER 1

## Lingo by Feature

This chapter lists various Macromedia Director MX features and the corresponding Lingo elements that you can use to implement those features.

### Accessibility

These terms are useful for making movies accessible to the disabled:

#### Text-to-speech

---

<code>voiceCount()</code>	<code>voiceSet()</code>
<code>voiceGet()</code>	<code>voiceSetPitch()</code>
<code>voiceGetPitch()</code>	<code>voiceSetRate()</code>
<code>voiceGetRate()</code>	<code>voiceSetVolume()</code>
<code>voiceGetVolume()</code>	<code>voiceSpeak()</code>
<code>voiceInitialize()</code>	<code>voiceState()</code>
<code>voicePause()</code>	<code>voiceStop()</code>
<code>voiceResume()</code>	<code>voiceWordPos()</code>

---

#### Keyboard navigation

---

<code>autoTab</code>	<code>selection</code> (cast property)
<code>hilite</code> (command)	<code>selectedText</code>
<code>keyboardFocusSprite</code>	<code>selEnd</code>
<code>selection</code> (text cast member property)	<code>selStart</code>
<code>selection()</code> (function)	

---

## Animated GIFs

These terms are useful for working with animated GIFs:

<code>directToStage</code>	<code>pause</code> (movie playback)
<code>frameRate</code>	<code>playBackMode</code>
<code>linked</code>	<code>resume sprite</code>
<code>moviePath</code>	<code>rewind sprite</code>

## Animation

These terms are useful for creating animation with Lingo:

<code>blend</code>	<code>locV</code>
<code>ink</code>	<code>member</code> (sprite property)
<code>loc</code>	<code>regPoint</code>
<code>locH</code>	<code>tweened</code>

## Behaviors

The terms in this section are useful for authoring behaviors and using behaviors while the movie plays.

### Authoring behaviors

Use these terms to set up behaviors and the behavior's Parameters dialog box:

<code>ancestor</code>	<code>on getBehaviorDescription</code>
<code>on runPropertyDialog</code>	<code>on getPropertyDescriptionList</code>
<code>on getBehaviorTooltip</code>	<code>property</code>
<code>on isOKToAttach</code>	

### Sending messages to behaviors

Use these commands to send messages to behaviors attached to sprites:

<code>call</code>	<code>sendSprite</code>
<code>callAncestor</code>	<code>sendAllSprites</code>

### Identifying behaviors

Use these terms to identify the behaviors attached to sprites:

<code>currentSpriteNum</code>	<code>scriptInstanceList</code>
<code>me</code>	<code>spriteNum</code>



## Bitmaps

The terms in this section are useful for working with bitmaps.

### Bitmap properties

Use these terms to check and set bitmap properties:

---

<code>alphaThreshold</code>	<code>foreColor</code>
<code>backColor</code>	<code>palette</code>
<code>blend</code>	<code>picture</code> (cast member property)
<code>depth</code>	<code>pictureP()</code>
<code>dither</code>	<code>rect</code> (member)
<code>trimWhiteSpace</code> (property)	<code>imageCompression</code>
<code>imageQuality</code>	<code>movieImageCompression</code>
<code>movieImageQuality</code>	

---

### Alpha channel

Use these terms to control alpha channel effects:

---

<code>alphaThreshold</code>	<code>dither</code>
<code>depth</code>	<code>useAlpha</code>
<code>createMask()</code>	<code>createMatte()</code>
<code>extractAlpha()</code>	<code>setAlpha()</code>

---

### Image objects

Use these terms to create and control image objects:

---

<code>copyPixels()</code>	<code>fill()</code>
<code>crop()</code> (image object command)	<code>image</code>
<code>draw()</code>	<code>image()</code>
<code>duplicate()</code> (image function)	<code>rect</code> (image)
<code>getPixel()</code>	<code>setPixel()</code>

---

## Buttons

See Buttons and check boxes in the Interface Elements section.

## Cast members

The terms in this section are useful for working with cast members.

### Creating cast members

Use `importFileInto` and `new()` to create cast members.

### Authoring

Use `duplicate member`, `erase member`, and `pasteClipBoardInto` to work with cast members during authoring.

### Graphic cast members

Use these terms to check and set the images assigned to graphic cast members:

---

<code>center</code>	<code>palette</code>
<code>crop (cast member property)</code>	<code>picture (cast member property)</code>
<code>depth</code>	<code>pictureP()</code>
<code>media</code>	<code>regPoint</code>

---

### General cast member properties

Use these terms to check and set cast member properties:

---

<code>fileName (cast member property)</code>	<code>number (cast member property)</code>
<code>media</code>	<code>preLoadMode</code>
<code>modified</code>	<code>type (cast member property)</code>
<code>name (cast member property)</code>	<code>URL</code>

---

### Graphic cast member dimensions

Use `height`, `rect (member)`, and `width` to check and set dimensions for graphic cast members.

## Casts

The terms in this section are useful for working with casts.

### Loading casts

Use `preLoadMode` to check and set when Director preloads a cast.

### Cast properties

Use these terms to specify cast properties:

---

<code>castLib</code>	<code>number (cast property)</code>
<code>fileName (cast property)</code>	<code>number (system property)</code>
<code>name (cast property)</code>	

---

## Cast management

Use these terms to manage casts:

---

activeCastLib	number of members
duplicate member	pasteClipboardInto
erase member	save castLib
findEmpty()	selection (cast property)
move member	

---

## Computer and operating system

Use these terms to check and control the computer:

---

beep	freeBlock()
beepOn	freeBytes()
cpuHogTicks	maxInteger
emulateMultiButtonMouse	multiSound
floatPrecision	romanLingo

---

## Operating system control

Use restart and shutDown to control the operating system.

## Data types

These terms are useful for specifying data types:

---

# (symbol)	string()
float()	stringP()
floatP()	symbol()
integer()	symbolP()
integerP()	VOID
objectP()	voidP()

---

## Digital video

These terms are useful for working with AVI and QuickTime digital video:

---

controller	trackNextSampleTime
digitalVideoTimeScale	trackPreviousKeyTime
digitalVideoType	trackPreviousSampleTime
directToStage	trackStartTime (sprite property)
duration	trackStartTime (cast member property)
frameRate	trackStopTime (sprite property)
loop (cast member property)	trackStopTime (cast member property)
movieRate	trackText

---

movieTime	trackType (cast member property)
pausedAtStart (Flash, digital video)	trackType (sprite property)
quickTimeVersion()	trackCount (cast member property)
timeScale	trackCount (sprite property)
trackEnabled	video (QuickTime, AVI)
trackNextKeyTime	videoForWindowsPresent

## QuickTime

Use these terms to work with QuickTime:

enableHotSpot	nodeType
fieldOfView	nudge
getHotSpotRect()	pan (QTVR property)
hotSpotExitCallback	ptToHotSpotID()
hotSpotEnterCallback	quickTimeVersion()
invertMask	rotation
isVRMovie	scale
loopBounds	swing()
mask	staticQuality
motionQuality	tilt
mouseLevel	translation
node	triggerCallback
nodeEnterCallback	warpMode
nodeExitCallback	

## RealMedia video

Use these terms to with RealMedia video:

audio (RealMedia)	play
currentTime (RealMedia)	realPlayerNativeAudio()
displayRealLogo	realPlayerPromptToInstall()
duration (RealMedia)	realPlayerVersion()
image (RealMedia)	seek
lastError	soundChannel (RealMedia)
mediaStatus	state (RealMedia)
password	stop (RealMedia)
pause (RealMedia)	userName (RealMedia)
pausedAtStart (RealMedia)	video (RealMedia)
percentBuffered	

## Events

Use these event handlers for Lingo that runs when a specific event occurs:

---

<code>activeCastLib</code>	<code>on moveWindow</code>
<code>close window</code>	<code>on mouseWithin</code>
<code>on cuePassed</code>	<code>open window</code>
<code>on deactivateWindow</code>	<code>on prepareFrame</code>
<code>on enterFrame</code>	<code>on prepareMovie</code>
<code>on EvalScript</code>	<code>on resizeWindow</code>
<code>on exitFrame</code>	<code>on mouseUpOutside</code>
<code>on idle</code>	<code>on rightMouseDown (event handler)</code>
<code>on keyDown</code>	<code>on rightMouseUp (event handler)</code>
<code>on keyUp</code>	<code>on startMovie</code>
<code>on mouseDown (event handler)</code>	<code>on stepFrame</code>
<code>on mouseEnter</code>	<code>on streamStatus</code>
<code>on mouseLeave</code>	<code>on timeOut</code>
<code>on mouseUp (event handler)</code>	<code>on zoomWindow</code>
<code>on stopMovie</code>	<code>on beginSprite</code>
<code>on endSprite</code>	<code>on hyperlinkClicked</code>

---

Use the `pass` and `stopEvent` commands to override the way that Director passes messages along the message hierarchy.

## External files

The terms in this section are useful for working with external files.

### Path names and filenames

Use these terms to check and set path names and filenames:

---

<code>@ (pathname)</code>	<code>getNthFileNameInFolder()</code>
<code>applicationPath</code>	<code>moviePath</code>
<code>fileName (cast property)</code>	<code>searchCurrentFolder</code>
<code>fileName (cast member property)</code>	<code>URL</code>

---

### Obtaining external media

Use these terms to obtain external media:

---

<code>downloadNetThing</code>	<code>preloadNetThing()</code>
<code>importFileInto</code>	

---

## Managing external files

Use these terms to manage external files:

---

<code>closeXlib</code>	<code>showXlib</code>
<code>open</code>	<code>sound playFile</code>
<code>openXlib</code>	

---

## Flash

These terms are useful for working with Flash cast members:

---

<code>actionsEnabled</code>	<code>originV</code>
<code>broadcastProps</code>	<code>pathName</code> (movie property)
<code>bufferSize</code>	<code>pausedAtStart</code> (Flash, digital video)
<code>buttonsEnabled</code>	<code>percentStreamed</code>
<code>bytesStreamed</code>	<code>play</code>
<code>callFrame()</code>	<code>playBackMode</code>
<code>centerRegPoint</code>	<code>playing</code>
<code>clearError</code>	<code>posterFrame</code>
<code>clickMode</code>	<code>print()</code>
<code>defaultRect</code>	<code>printAsBitmap()</code>
<code>defaultRectMode</code>	<code>quality</code>
<code>directToStage</code>	<code>rewind sprite</code>
<code>endTellTarget()</code> See <code>tellTarget()</code>	<code>rotation</code>
<code>eventPassMode</code>	<code>scale</code>
<code>findLabel()</code>	<code>scaleMode</code>
<code>fixedRate</code>	<code>on sendXML</code>
<code>flashRect</code>	<code>setCallback()</code>
<code>flashToStage()</code>	<code>settingsPanel()</code>
<code>frame()</code> (function)	<code>setFlashProperty()</code>
<code>frame</code> (sprite property)	<code>setVariable()</code>
<code>frameCount</code>	<code>showProps()</code>
<code>frameRate</code>	<code>sound</code>
<code>frameReady()</code>	<code>the soundMixMedia</code>
<code>getError()</code>	<code>sourceFileName</code>
<code>getFlashProperty()</code>	<code>stageToFlash()</code>
<code>getFrameLabel()</code>	<code>state</code> (Flash, SWA)
<code>getVariable()</code>	<code>static</code>
<code>goToFrame</code>	<code>stop</code> (Flash)

---

hitTest()	stream
hold	streamMode
imageEnabled	streamSize
linked	tellTarget()
loop (keyword)	URL
mouseoverButton	viewH
newObject()	viewPoint
obeyScoreRotation	viewScale
originH	viewV
originMode	volume (cast member property)
originPoint	

These terms are useful for working with global Flash objects, which do not require a Flash cast member:

clearAsObjects()	setCallback()
newObject()	

## Frames

The Lingo terms in this section let you work with frames.

### Frame events

Use the on enterFrame, on exitFrame, and on prepareFrame event handlers to contain Lingo that runs at specific events within a frame.

### Frame properties

Use these Lingo terms to check and set frame properties:

frameLabel	frameTempo
framePalette	frameTransition
frameScript	label()
frameSound1	labelList
frameSound2	marker()
markerList	

## Interface elements

The Lingo terms in this section are useful for working with interface elements.

## Menus

Use these terms to create menus:

enabled	name (menu item property)
installMenu	number (menu items)
menu	number (menus)
name (menu property)	script

## Buttons and check boxes

Use these terms to specify buttons and check boxes:

alert	checkBoxType
buttonStyle	checkMark
buttonType	hilite (cast member property)
checkBoxAccess	

## Keys

The Lingo terms in this section are related to using the keyboard.

### Identifying keys

Use these terms to identify keys:

charToNum()	keyPressed()
commandDown	mouseChar
controlDown	numToChar()
key()	optionDown
keyCode()	shiftDown

### Keyboard interaction

Use `keyPressed()`, `lastEvent()`, and `lastKey` to detect what the user types at the keyboard.

### Keyboard events

Use these terms to set up handlers that respond to pressing keys:

on keyDown	keyDownScript
on keyUp	keyUpScript
flushInputEvents	

## Lingo

The Lingo terms in this section are important language elements that you use to construct scripts.



## Boolean values

Use these terms to test whether a condition exists:

- FALSE (0 is the numerical equivalent of FALSE).
- TRUE (1 is the numerical equivalent of TRUE).
- not
- or

## Script control

Use these terms to control how a script executes:

---

abort	pass
do	result
exit	scriptsEnabled
halt	scriptText
nothing	stopEvent

---

## Code structures

Use if to create if...then statements.

Use case, end case, and otherwise in case statements.

Use these terms for repeat loops:

---

end repeat	repeat with
exit repeat	repeat with...down to
next repeat	repeat with...in list
repeat while	

---

## Syntax elements

Use these terms as part of Lingo's syntax:

---

# (symbol)	member (keyword)
" (string)	of
↪ (continuation)	or
-- (comment)	property
() (parentheses)	sprite
castLib	the
end	window
global	

---

## Lists

The terms in this section are useful for working with lists.

### Creating lists

Use `[ ]` (list), `duplicate()` (list function), or `list()` to create a list.

### Adding list items

Use these terms to add items to a list:

<code>[ ]</code> (bracket access)	<code>addVertex</code>
<code>add</code>	<code>append</code>
<code>addVertex</code>	

### Deleting list items

Use these terms to delete items from a list:

<code>deleteAll</code>	<code>deleteOne</code>
<code>deleteAt</code>	<code>deleteProp</code>

### Retrieving values from a list

Use these terms to retrieve values from a list:

<code>[ ]</code> (bracket access)	<code>getOne()</code>
<code>deleteProp</code>	<code>getPos()</code>
<code>deleteProp</code>	<code>getProp()</code>
<code>getLast()</code>	<code>getPropAt()</code>

### Getting information about lists

Use these terms to get information about lists:

<code>count()</code>	<code>max()</code>
<code>findPos</code>	<code>min</code>
<code>findPosNear</code>	<code>param()</code>
<code>ilk()</code>	<code>paramCount()</code>
<code>listP()</code>	

### Setting values in a list

Use these terms to set values in a list:

<code>[ ]</code> (bracket access)	<code>setAt</code>
<code>setaProp</code>	<code>setProp</code>

## Media synchronization

Use these terms to synchronize animation and sound:

---

<code>cuePointNames</code>	<code>on cuePassed</code>
<code>cuePointTimes</code>	<code>isPastCuePoint()</code>
<code>mostRecentCuePoint</code>	

---

## Memory management

The terms in this section are useful for determining memory requirements and controlling when the movie loads and unloads cast members.

### Idle events

Use the `on idle` event handler for Lingo that runs when the movie is idle.

### Idle loading

Use these terms to control idle loading:

---

<code>cancelIdleLoad</code>	<code>idleLoadPeriod</code>
<code>finishIdleLoad</code>	<code>idleLoadTag</code>
<code>idleHandlerPeriod</code>	<code>idleReadChunkSize</code>
<code>idleLoadDone()</code>	<code>netThrottleTicks</code>

---

## Preloading and querying media

Use these terms to load media into memory and check whether media are available:

---

<code>frameReady()</code>	<code>preloadNetThing()</code>
<code>loaded</code>	<code>preloadMember</code>
<code>mediaReady</code>	<code>preloadMovie</code>
<code>preload (command)</code>	<code>preloadRAM</code>
<code>preload (cast member property)</code>	<code>purgePriority</code>
<code>preloadBuffer member</code>	<code>unload</code>
<code>preloadEventAbort</code>	<code>unloadMember</code>
<code>preloadMode</code>	<code>unloadMovie</code>

---

## Available memory

Use these terms to check how much memory is available:

---

<code>freeBlock()</code>	<code>movieFileFreeSize</code>
<code>freeBytes()</code>	<code>movieFileSize</code>
<code>memorySize</code>	

---

## Memory requirements

Use `ramNeeded()` and `size` to determine how much memory required for a cast member or a range of frames.

## Menus

See Menus in the Interface elements section.

## Message window

Use these terms to work in the Message window:

---

<code>put</code>	<code>traceLoad</code>
<code>showXlib</code>	<code>traceLogFile</code>
<code>trace</code>	<code>appMinimize</code>

---

## Monitor

Use `colorDepth`, `deskTopRectList`, and `switchColorDepth` to check and control the monitor.

## Mouse interaction

The terms in this section are useful for Lingo related to using the mouse.

### Mouse clicks

Use these terms to detect what the user does with the mouse:

---

<code>clickOn</code>	<code>mouseLine</code>
<code>doubleClick</code>	<code>mouseLoc</code>
<code>emulateMultiButtonMouse</code>	<code>mouseMember</code>
<code>lastClick()</code>	<code>mouseOverButton</code>
<code>lastEvent()</code>	<code>on mouseUp (event handler)</code>
<code>lastRoll</code>	<code>mouseV</code>
<code>mouseChar</code>	<code>mouseWord</code>
<code>on mouseDown (event handler)</code>	<code>on rightMouseDown (event handler)</code>
<code>mouseH</code>	<code>on rightMouseUp (event handler)</code>
<code>mouseItem</code>	<code>rollover()</code>
<code>mouseLevel</code>	<code>stillDown</code>

---

## Mouse events

Use these terms to set up handlers that respond to mouse events:

---

<code>mouseDownScript</code>	<code>on mouseUp</code> (event handler)
<code>mouseUpScript</code>	<code>on mouseUpOutside</code>
<code>on mouseDown</code> (event handler)	<code>on mouseWithin</code>
<code>on mouseEnter</code>	<code>on rightMouseDown</code> (event handler)
<code>on mouseLeave</code>	<code>on rightMouseUp</code> (event handler)

---

## Cursor control

Use `cursor` (command), `cursor` (sprite property), and `cursorSize` to control the pointer (cursor).

## Movies in a window

The terms in this section are useful for working with movies in a window.

### Movie in a window events

Use these event handlers to contain Lingo that you want to run in response to events in a movie in a window:

---

<code>activeCastLib</code>	<code>on openWindow</code>
<code>on closeWindow</code>	<code>on resizeWindow</code>
<code>on moveWindow</code>	<code>on zoomWindow</code>

---

### Opening and closing movies in a window

Use these terms for opening and closing windows:

---

<code>close window</code>	<code>open window</code>
<code>forget</code>	<code>windowList</code>

---

## Window appearance

Use these terms to check and set the appearance of a movie's window:

---

<code>drawRect</code>	<code>sourceRect</code>
<code>fileName</code> (window property)	<code>tell</code>
<code>frontWindow</code>	<code>title</code>
<code>modal</code>	<code>titleVisible</code>
<code>moveToBack</code>	<code>visible</code> (window property)
<code>moveToFront</code>	<code>windowPresent()</code>
<code>name</code> (window property)	<code>windowType</code>
<code>rect</code> (window)	<code>appMinimize</code>

---

## Communication between movies

Use the `tell` command to send messages between movies.

## Movies

The terms in this section are useful for managing movies.

### Stopping movies

Use these terms to stop or quit the movie or projector:

---

<code>exitLock</code>	<code>quit</code>
<code>halt</code>	<code>restart</code>
<code>pauseState</code>	<code>shutDown</code>

---

### Movie information

Use these terms to obtain information about the movie and the movie's environment:

---

<code>environment</code>	<code>moviePath</code>
<code>lastFrame</code>	<code>number</code> (system property)
<code>movie</code>	<code>runMode</code>
<code>movieFileFreeSize</code>	<code>safePlayer</code>
<code>movieFileSize</code>	<code>version</code>
<code>movieName</code>	<code>movieFileVersion</code>

---

### Source control

Use these terms to manage Director projects being worked on by more than one person:

---

<code>comments</code>	<code>creationDate</code>
<code>modifiedBy</code>	<code>modifiedDate</code>
<code>linkAs()</code>	<code>seconds</code>

---

### Saving movies

Use `saveMovie` and `updateMovieEnabled` to save changes to a movie.

### Error checking

Use the `alertHook` event to post alerts that describe errors in a projector.

### Movie events

Use the `on prepareMovie`, `on startMovie`, and `on stopMovie` event handlers for Lingo that responds to movie events.

## Multiuser server

Director MX users should use Macromedia Flash Communication Server MX for communication among Director movies and with application servers. For more information about using Flash Communication Server MX, see Using Flash Communication Server MX in Using Director.

## Navigation

Use these terms to jump to different locations:

---

delay	goToFrame
go	gotoNetMovie
go loop	gotoNetPage
go next	play
go previous	play done

---

## Network Lingo

The terms in this section are useful for working with the network.

### Downloading and streaming media

Use these terms to obtain or stream media from the network:

---

downloadNetThing (For projectors and authoring only)	gotoNetPage
getNetText()	postNetText
gotoNetMovie	preloadNetThing()

---

### Checking availability

Use `frameReady()` and `mediaReady` to check whether specific media are completely downloaded.

### Using network operations

Use these terms to check the progress of a network operation or get information regarding network media:

---

getStreamStatus()	netMIME()
getLatestNetID	netTextResult()
netAbort	on streamStatus
netDone()	proxyServer
netError()	tellStreamStatus()
netPresent	URLEncode
netLastModDate()	

---

## Working with the local computer

Use these terms to work with the user's computer:

---

<code>browserName()</code>	<code>clearCache</code> (For projectors and authoring only)
<code>cacheDocVerify()</code> (For projectors and authoring only)	<code>getPref()</code>
<code>cacheSize()</code> (For projectors and authoring only)	<code>setPref</code>

---

## Browsers

Use on `EvalScript`, `externalEvent`, and `netStatus` to interact with browsers. For additional information about browser scripting using languages such as JavaScript, see “Shockwave Publishing” on the Director Support Center at [www.macromedia.com/support/director/internet.html](http://www.macromedia.com/support/director/internet.html).

## Accessing EMBED and OBJECT tag parameters

Use `externalParamCount()`, `externalParamName()`, and `externalParamValue()` to access EMBED and OBJECT parameter tags:

## Operators

The terms in this section are operators available in Lingo.

### Math operators

Use these terms for math statements:

---

<code>*</code> (multiplication)	<code>&lt;&gt;</code> (not equal)
<code>/</code> (division)	<code>&gt;</code> (greater than)
<code>+</code> (addition)	<code>&gt;=</code> (greater than or equal to)
<code>-</code> (minus)	<code>&lt;</code> (less than)
<code>=</code> (equals)	<code>&lt;=</code> (less than or equal to)

---

### Comparison operators

Use `and`, `not`, and `or` to compare expressions.

## Palettes and color

Use these terms to check and set palettes for movies and for cast members:

---

<code>color()</code>	<code>paletteMapping</code>
<code>depth</code>	<code>puppetPalette</code>
<code>palette</code>	<code>rgb()</code>

---



## Parent scripts

Use these terms to work with parent scripts and child objects:

---

actorList	property
ancestor	on stepFrame
new()	handler()
handlers()	rawNew()

---

## Points and rectangles

These terms are useful for checking and setting points and rectangles.

---

inflate	quad
inside()	rect (camera)
intersect()	rect (sprite)
map()	sourceRect
offset() (rectangle function)	union()
point()	

---

For Lingo that controls a sprite's bounding rectangle, see [Sprite dimensions](#).

## Projectors

These terms are useful for working with projectors:

---

alertHook	platform
environment	runMode
editShortCutsEnabled	

---

## Puppets

Use these terms to control the puppet property of sprites and effects channels:

---

puppet	puppetTempo
puppetPalette	puppetTransition
puppetSound	updateStage
puppetSprite	

---

## Random numbers

Use `random()` and `randomSeed` to generate random numbers.

## Score

The following terms let you work with the Score.

### Score properties

Use `lastFrame`, `score`, and `scoreSelection` to work with the movie's Score.

### Score generation

Use these terms to create Score content from Lingo:

<code>beginRecording</code>	<code>scoreSelection</code>
<code>clearFrame</code>	<code>scriptNum</code>
<code>deleteFrame</code>	<code>scriptType</code>
<code>duplicateFrame</code>	<code>tweened</code>
<code>endRecording</code>	<code>updateFrame</code>
<code>insertFrame</code>	<code>updateLock</code>
<code>scoreColor</code>	

## Shapes

Use these Lingo terms to work with shapes:

<code>filled</code>	<code>pattern</code>
<code>lineDirection</code>	<code>shapeType</code>
<code>lineSize</code>	

## Shockwave audio

Use these terms to check, stream, and play Shockwave audio sounds:

<code>bitRate</code>	<code>play</code> member
<code>bitsPerSample</code>	<code>preLoadBuffer</code> member
<code>copyrightInfo</code>	<code>preLoadTime</code>
<code>duration</code>	<code>sampleRate</code>
<code>getError()</code>	<code>soundChannel</code> (SWA)
<code>getErrorString()</code>	<code>state</code> (Flash, SWA)
<code>numChannels</code>	<code>stop</code> member
<code>pause</code> (movie playback)	<code>streamName</code>
<code>percentPlayed</code>	<code>URL</code>
<code>percentStreamed</code>	<code>volume</code> (cast member property)

## Sound

The terms in this section are useful for playing sounds.

### Sound information

Use these terms to get information about a sound:

---

<code>channelCount</code>	<code>soundEnabled</code>
<code>sound</code>	<code>volume</code> (sprite property)
<code>soundBusy()</code>	<code>isBusy()</code>
<code>sampleCount</code>	<code>status</code>

---

### Playing sound

Use these terms to control how sound plays:

---

<code>puppetSound</code>	<code>sound fadeOut</code>
<code>sound close</code>	<code>sound playFile</code>
<code>sound fadeIn</code>	<code>sound stop</code>
<code>breakLoop()</code>	<code>elapsedTime</code>
<code>endTime</code>	<code>fadeIn()</code>
<code>fadeOut()</code>	<code>fadeTo()</code>
<code>getPlaylist()</code>	<code>setPlaylist()</code>
<code>loopCount</code>	<code>loopEndTime</code>
<code>loopStartTime</code>	<code>loopsRemaining</code>
<code>member</code> (sound property)	<code>pan</code> (sound property)
<code>pause()</code> (sound playback)	<code>playNext()</code>
<code>queue()</code>	<code>rewind()</code>
<code>stop()</code> (sound)	<code>play()</code> (sound)

---

### RealMedia sound

See Digital video.

## Sprites

The Lingo terms in this section are for sprites.

### Sprite events

Use the `on beginSprite` and `on endSprite` event handlers to contain Lingo that you want to run when a sprite begins or ends.

### Assigning cast members to sprites

Use `castLibNum`, `member` (sprite property), or `memberNum` to specify a sprite's cast member.

## Rotating sprites

Use the `rotation` sprite property to rotate sprites.

## Dragging sprites

Use these terms to set how the user can drag sprites:

---

<code>constrainH()</code>	<code>moveableSprite</code>
<code>constrainV()</code>	<code>sprite...intersects</code>
<code>constraint</code>	<code>sprite...within</code>

---

## Sprites and Lingo

Use these terms to manage how Lingo controls sprites:

---

<code>puppetSprite</code>	<code>spriteNum</code>
<code>puppet</code>	<code>sendSprite</code>
<code>scriptNum</code>	<code>sendAllSprites</code>
<code>scriptInstanceList</code>	

---

## Drawing sprites on the Stage

Use these terms to control how Director draws a sprite on the Stage:

---

<code>blend</code>	<code>skew</code>
<code>flipH</code>	<code>trails</code>
<code>flipV</code>	<code>tweened</code>
<code>ink</code>	<code>updateStage</code>
<code>quad</code>	<code>visible</code> (sprite property)
<code>rotation</code>	

---

## Sprite dimensions

Use these terms to check and set the size of a sprite's bounding rectangle:

---

<code>bottom</code>	<code>right</code>
<code>height</code>	<code>top</code>
<code>left</code>	<code>width</code>
<code>quad</code>	<code>zoomBox</code>

---

You can also manipulate a sprite's bounding rectangle with Lingo for rectangles. See [Points and rectangles](#).

## Sprite locations

Use the `loc`, `locH`, and `locV` sprite properties to check and set sprite locations.

## Sprite color

Use these terms to check and set a sprite's color:

---

<code>backColor</code>	<code>color</code> (sprite and cast member property)
<code>bgColor</code>	<code>foreColor</code>

---

## Stage

These terms are useful for controlling the Stage and determining its size and location:

---

<code>centerStage</code>	<code>stageColor</code>
<code>fixStageSize</code>	<code>stageLeft</code>
<code>picture</code> (window property)	<code>stageRight</code>
<code>stage</code>	<code>stageTop</code>
<code>stageBottom</code>	<code>updateStage</code>

---

## Tempo

Use the `puppetTempo` command to control a movie's tempo.

## Text

The terms in this section are useful for working with text, strings, and fields.

### Manipulating strings

Use these terms to manipulate strings:

---

<code>&amp;</code> (concatenator)	<code>put...before</code>
<code>&amp;&amp;</code> (concatenator)	<code>put...into</code>
<code>delete</code>	<code>string()</code>
<code>hilite</code> (cast member property)	<code>stringP()</code>
<code>put...after</code>	<code>text</code>

---

### Chunk expressions

Use these terms to identify chunks of text:

---

<code>char...of</code>	<code>number</code> (words)
<code>chars()</code>	<code>offset()</code> (string function)
<code>contains</code>	<code>paragraph</code>
<code>EMPTY</code>	<code>ref</code>
<code>item...of</code>	<code>selection</code> (text cast member property)
<code>itemDelimiter</code>	<code>selectedText</code>
<code>last()</code>	<code>selEnd</code> (fields only)
<code>length()</code>	<code>selStart</code> (fields only)

---

line...of	string()
number (characters)	stringP()
number (items)	value()
number (lines)	word...of

## Editable text

Use the `editable` property to specify whether text is editable.

## Shocked fonts

Use these terms to include Shocked fonts with downloaded text:

<code>recordFont</code>	<code>bitmapSizes</code>
<code>originalFont</code>	<code>characterSet</code>

## Character formatting

Use these terms to format text:

<code>backColor</code>	<code>fontf</code>
<code>bgColor</code>	<code>fontSize</code>
<code>charSpacing</code>	<code>fontStyle</code>
<code>color()</code>	<code>foreColor</code>
<code>dropShadow</code>	

## Paragraph formatting

Use these terms to format paragraphs:

<code>alignment</code>	<code>rightIndent</code>
<code>bottomSpacing</code>	<code>tabCount</code>
<code>firstIndent</code>	<code>tabs</code>
<code>fixedLineSpace</code>	<code>top (3D)</code>
<code>leftIndent</code>	<code>wordWrap</code>
<code>margin</code>	

## Text cast member properties

Use these terms to work with the entire text content of a text cast member:

<code>antiAlias</code>	<code>kerning</code>
<code>antiAliasThreshold</code>	<code>kerningThreshold</code>
<code>autoTab</code>	<code>picture (cast member property)</code>
<code>HTML</code>	<code>RTF</code>

Lingo that applies to chunk expressions is also available to the text within a text cast member.

## Mouse pointer position in text

Use these terms to detect where the mouse pointer is within text:

<code>pointInHyperlink()</code>	<code>pointToParagraph()</code>
<code>pointToChar()</code>	<code>pointToWord()</code>
<code>pointToItem()</code>	

## Text boxes for field cast members

Use these terms to set up the box for a field cast member:

<code>border</code>	<code>lineHeight()</code> (function)
<code>boxType</code>	<code>lineHeight</code> (cast member property)
<code>lineCount</code>	<code>pageHeight</code>

## Scrolling text

Use these terms to work with scrolling text:

<code>linePosToLocV()</code>	<code>scrollByLine</code>
<code>locToCharPos()</code>	<code>scrollByPage</code>
<code>locVToLinePos()</code>	<code>scrollTop</code>

## Constants

Use these terms to work with constants:

<code>BACKSPACE</code>	<code>RETURN</code> (constant)
<code>EMPTY</code>	<code>VOID</code>
<code>ENTER</code>	

## Time

The terms in this section are useful for working with time.

### Current date and time

Use these terms to determine the current date and time:

<code>abbr</code> , <code>abbrev</code> , <code>abbreviated</code>	<code>short</code>
<code>date()</code> (system clock)	<code>systemDate</code>
<code>long</code>	

## Measuring time

Use these terms to measure time in a movie:

<code>framesToHMS()</code>	<code>ticks</code>
<code>HMSToFrames()</code>	<code>time()</code>
<code>milliseconds</code>	<code>timer</code>
<code>startTimer</code>	

## Timeouts

Use these terms to handle timeouts:

<code>timeoutKeyDown</code>	<code>timeoutMouse</code>
<code>timeoutLapsed</code>	<code>timeoutPlay</code>
<code>timeoutLength</code>	<code>timeoutScript</code>
<code>name (timeout property)</code>	<code>period</code>
<code>persistent</code>	<code>target</code>
<code>time()</code>	<code>timeout()</code>
<code>timeoutHandler</code>	<code>timeoutList</code>

## Transitions

Use these terms to work with transitions:

<code>changeArea</code>	<code>puppetTransition</code>
<code>chunkSize</code>	<code>transitionType</code>
<code>duration</code>	

## Variables

The terms in this section are useful for creating and changing variables:

### Creating variables

Use these terms to create variables:

<code>= (equals)</code>	<code>property</code>
<code>global</code>	

### Testing and changing variables

Use these terms to check and change the values assigned to variables:

<code>= (equals)</code>	<code>put</code>
<code>clearGlobals</code>	<code>set...to, set...=</code>
<code>globals</code>	<code>showGlobals</code>
<code>ilk()</code>	<code>showLocals</code>



## Vector shapes

Use these Lingo terms to work with vector shapes:

---

<code>addVertex</code>	<code>gradientType</code>
<code>antiAlias</code>	<code>imageEnabled</code>
<code>backgroundColor</code>	<code>moveVertex()</code>
<code>broadcastProps</code>	<code>moveVertexHandle()</code>
<code>centerRegPoint</code>	<code>originH</code>
<code>closed</code>	<code>originMode</code>
<code>defaultRect</code>	<code>originPoint</code>
<code>defaultRectMode</code>	<code>originV</code>
<code>deleteVertex()</code>	<code>rotation</code>
<code>directToStage</code>	<code>scale</code>
<code>endColor</code>	<code>scaleMode</code>
<code>fillColor</code>	<code>showProps()</code>
<code>fillCycles</code>	<code>skew</code>
<code>fillDirection</code>	<code>static</code>
<code>fillMode</code>	<code>strokeColor</code>
<code>fillOffset</code>	<code>strokeWidth</code>
<code>fillScale</code>	<code>vertexList</code>
<code>flashRect</code>	<code>viewPoint</code>
<code>flipH</code>	<code>viewScale</code>
<code>flipV</code>	<code>viewV</code>
<code>curve</code>	<code>newCurve()</code>
<code>regPointVertex</code>	

---

## XML parsing

The following Lingo is useful for XML parsing within Director.

---

<code>attributeName</code>	<code>ignoreWhiteSpace()</code>
<code>attributeValue</code>	<code>makeList()</code>
<code>child (XML)</code>	<code>makeSubList()</code>
<code>count()</code>	<code>name (XML property)</code>
<code>doneParsing()</code>	<code>parseString()</code>
<code>getError() (XML)</code>	<code>parseURL()</code>

---

## Xtra extensions

Use these terms to work with Xtra extensions:

movieXtraList	xtra
name (system property)	xtraList
number of xtras	xtras

# CHAPTER 2

## 3D Lingo by Feature

This chapter lists the various 3D features of Macromedia Director MX and the corresponding Lingo elements that you can use to implement those features.

### Animation

Use these terms to work with 3D animation. See also the lists of terms for the Keyframe player and Bones player modifiers.

---

<code>animationEnabled</code>	<code>pause() (3D)</code>
<code>autoblend</code>	<code>play() (3D)</code>
<code>blendTime</code>	<code>playing (3D)</code>
<code>cloneMotionFromCastmember</code>	<code>playlist</code>
<code>count</code>	<code>playNext() (3D)</code>
<code>currentLoopState</code>	<code>playRate</code>
<code>currentTime (3D)</code>	<code>positionReset</code>
<code>deleteMotion</code>	<code>queue() (3D)</code>
<code>lockTranslation</code>	<code>removeLast()</code>
<code>motion</code>	<code>rotationReset</code>
<code>name</code>	<code>type (motion)</code>
<code>newMotion()</code>	

---

### Anti-aliasing

Use these terms to work with anti-aliasing:

---

<code>antiAliasingEnabled</code>	<code>antiAliasingSupported</code>
----------------------------------	------------------------------------

---

## Backdrops and overlays

Use these terms to manipulate backdrops and overlays in 3D cast members:

---

<code>addBackdrop</code>	<code>regPoint</code> (3D)
<code>addOverlay</code>	<code>removeBackdrop</code>
<code>blend</code> (3D)	<code>removeOverlay</code>
<code>count</code>	<code>rotation</code> (backdrop and overlay)
<code>insertBackdrop</code>	<code>scale</code> (backdrop and overlay)
<code>insertOverlay</code>	<code>source</code>
<code>loc</code> (backdrop and overlay)	

---

## Bones player modifier

Use these terms to control the functionality of the Bones player modifier:

---

<code>autoblend</code>	<code>play()</code> (3D)
<code>blendTime</code>	<code>playing</code> (3D)
<code>bonesPlayer</code> (modifier)	<code>playlist</code>
<code>count</code>	<code>playNext()</code> (3D)
<code>currentTime</code> (3D)	<code>playRate</code>
<code>getBoneID</code>	<code>queue()</code> (3D)
<code>currentLoopState</code>	<code>removeLast()</code>
<code>getWorldTransform()</code>	<code>rootLock</code>
<code>lockTranslation</code>	<code>rotationReset</code>
<code>positionReset</code>	<code>transform</code> (property)
<code>pause()</code> (3D)	

---

## Cameras

Use these terms to work with cameras and camera properties:

---

<code>addCamera</code>	<code>orthoHeight</code>
<code>addToWorld</code>	<code>pointAt</code>
<code>autoCameraPosition</code>	<code>pointAtOrientation</code>
<code>boundingSphere</code>	<code>position</code> (transform)
<code>camera</code>	<code>projection</code>
<code>cameraCount()</code>	<code>projectionAngle</code>
<code>cameraPosition</code>	<code>rect</code> (camera)
<code>cameraRotation</code>	<code>removeFromWorld</code>
<code>clone</code>	<code>rootNode</code>
<code>cloneDeep</code>	<code>rotate</code>

---

<i>count</i>	<i>scale (transform)</i>
<i>deleteCamera</i>	<i>transform (property)</i>
<i>fieldOfView (3D)</i>	<i>translate</i>
<i>hither</i>	<i>userData</i>
<i>isInWorld()</i>	<i>worldPosition</i>
<i>name</i>	<i>yon</i>
<i>newCamera</i>	

## Child and parent nodes

Use these terms to control parent-child relationships between models:

<i>addChild</i>	<i>count</i>
<i>child</i>	<i>parent</i>

## Collision detection

These terms are useful for detecting and responding to collisions between models:

<i>collision (modifier)</i>	<i>pointOfContact</i>
<i>collisionData</i>	<i>registerForEvent()</i>
<i>collisionNormal</i>	<i>registerScript()</i>
<i>enabled (collision)</i>	<i>resolve</i>
<i>immovable</i>	<i>resolveA</i>
<i>window("Tool Panel").modal = FALSE</i>	<i>resolveB</i>
<i>modelA</i>	<i>setCollisionCallback()</i>
<i>modelB</i>	

## Creating and removing objects

Use these terms to create and remove objects:

<i>add (3D texture)</i>	<i>deleteShader</i>
<i>addBackdrop</i>	<i>deleteTexture</i>
<i>addModifier</i>	<i>duplicate</i>
<i>addOverlay</i>	<i>insertBackdrop</i>
<i>addToWorld</i>	<i>insertOverlay</i>
<i>camera</i>	<i>newLight</i>
<i>child</i>	<i>newMesh</i>
<i>clone</i>	<i>newModel</i>
<i>cloneDeep</i>	<i>newModelResource</i>
<i>cloneModelFromCastmember</i>	<i>newMotion()</i>

<code>cloneMotionFromCastmember</code>	<code>newShader</code>
<code>deleteCamera</code>	<code>newTexture</code>
<code>deleteGroup</code>	<code>removeModifier</code>
<code>deleteLight</code>	<code>removeBackdrop</code>
<code>deleteModel</code>	<code>removeFromWorld</code>
<code>deleteModelResource</code>	<code>removeOverlay</code>
<code>deleteMotion</code>	

## Fog

Use these terms to work with fog:

<code>color (fog)</code>	<code>far (fog)</code>
<code>decayMode</code>	<code>fog</code>
<code>enabled (fog)</code>	<code>near (fog)</code>

## Groups

Use these terms to work with groups:

<code>addChild</code>	<code>newGroup</code>
<code>addToWorld</code>	<code>pointAt</code>
<code>boundingSphere</code>	<code>pointAtOrientation</code>
<code>child</code>	<code>position (transform)</code>
<code>clone</code>	<code>removeFromWorld</code>
<code>cloneDeep</code>	<code>rotate</code>
<code>count</code>	<code>scale (transform)</code>
<code>deleteGroup</code>	<code>transform (property)</code>
<code>group</code>	<code>translate</code>
<code>isInWorld()</code>	<code>userData</code>
<code>name</code>	<code>worldPosition</code>

## Inker modifier

Use these terms to control the functionality of the Inker modifier:

<code>boundary</code>	<code>lineColor</code>
<code>creaseAngle</code>	<code>lineOffset</code>
<code>creases</code>	<code>silhouettes</code>
<code>inker (modifier)</code>	<code>useLineOffset</code>

## Keyframe player modifier

Use these terms to control the functionality of the Keyframe player modifier:

autoblend	playing (3D)
blendFactor	playlist
blendTime	playNext() (3D)
count	playRate
currentLoopState	positionReset
currentTime (3D)	queue() (3D)
keyframePlayer (modifier)	removeLast()
lockTranslation	rootLock
pause() (3D)	rotationReset
play() (3D)	update

## Level of detail modifier

Use these terms to control the functionality of the level of detail (LOD) modifier:

auto	level
bias	lod (modifier)

## Lights

Use these terms to work with lights and light properties:

addToWorld	pointAt
ambientColor	pointAtOrientation
attenuation	position (transform)
boundingSphere	removeFromWorld
color (light)	rotate
count	scale (transform)
clone	specular (light)
cloneDeep	spotAngle
deleteLight	spotDecay
directionalColor	transform (property)
directionalPreset	translate
isInWorld()	type (light)
light	userData
name	worldPosition
newLight	

## Mesh deform modifier

Use these terms to control the functionality of the mesh deform modifier:

<code>add (3D texture)</code>	<code>normalList</code>
<code>face</code>	<code>textureCoordinateList</code>
<code>mesh (property)</code>	<code>textureLayer</code>
<code>meshDeform (modifier)</code>	<code>vertexList (mesh deform)</code>
<code>neighbor</code>	

## Miscellaneous

<code>clearAtRender</code>	<code>resetWorld</code>
<code>clearValue</code>	<code>revertToWorldDefaults</code>
<code>directToStage</code>	<code>sendEvent</code>
<code>loadFile()</code>	<code>setCollisionCallback()</code>
<code>registerForEvent()</code>	<code>unregisterAllEvents</code>
<code>registerScript()</code>	<code>revertToWorldDefaults</code>

## Model resources

Use these terms to work with 3D model resources:

<code>count</code>	<code>newModelResource</code>
<code>deleteModelResource</code>	<code>resolution</code>
<code>modelResource</code>	<code>resource</code>
<code>name</code>	<code>type (model resource)</code>

## Models

Use these terms to work with 3D models:

<code>addToWorld</code>	<code>position (transform)</code>
<code>boundingSphere</code>	<code>removeFromWorld</code>
<code>clone</code>	<code>renderStyle</code>
<code>cloneDeep</code>	<code>resource</code>
<code>cloneModelFromCastmember</code>	<code>rotate</code>
<code>count</code>	<code>scale (transform)</code>
<code>deleteModel</code>	<code>shadowPercentage</code>
<code>isInWorld()</code>	<code>shaderList</code>
<code>model</code>	<code>transform (property)</code>
<code>modifier</code>	<code>translate</code>



name	userData
newModel	visibility
pointAt	worldPosition
pointAtOrientation	

## Modifiers

These terms are useful for applying modifiers to models and model resources. See the name of the specific modifier you are using for a list of terms that work with that modifier.

addModifier	modifiers
count	removeModifier
modifier	

## Movie and system properties

Use these terms to determine the 3D capabilities of the playback computer:

active3dRenderer	getRendererServices()
colorBufferDepth	preferred3DRenderer
depthBufferDepth	renderer
getHardwareInfo()	rendererDeviceList

## Nodes

Use these terms to manage nodes. A node is any object that exists in the world, including lights, cameras, models, and groups.

addToWorld	isInWorld()
clone	name
cloneDeep	removeFromWorld
count	userData

## Particle systems

See Primitives.

## Picking

See Selecting models.

## Primitives

The following sections list the terms used to work with each type of primitive.

Use the `primitives` property to determine which types of primitives are supported by the current 3D renderer.

### Boxes

Use these terms to control properties of 3D boxes:

---

<code>back</code>	<code>length (3D)</code>
<code>bottom (3D)</code>	<code>lengthVertices</code>
<code>front</code>	<code>right (3D)</code>
<code>height (3D)</code>	<code>top (3D)</code>
<code>heightVertices</code>	<code>width (3D)</code>
<code>left (3D)</code>	<code>widthVertices</code>

---

### Cylinders

Use these terms to control properties of 3D cylinders:

---

<code>bottomCap</code>	<code>resolution</code>
<code>bottomRadius</code>	<code>state (3D)</code>
<code>endAngle</code>	<code>topCap</code>
<code>height (3D)</code>	<code>topRadius</code>
<code>numSegments</code>	

---

### Meshes

Use these terms to control properties of 3D meshes:

---

<code>build()</code>	<code>normalList</code>
<code>colorList</code>	<code>shadowPercentage</code>
<code>count</code>	<code>textureCoordinateList</code>
<code>face</code>	<code>textureCoordinates</code>
<code>generateNormals()</code>	<code>vertexList (mesh deform)</code>
<code>newMesh</code>	

---

### Particle systems

Use these terms to control properties of 3D particle systems:

---

<code>angle</code>	<code>minSpeed</code>
<code>blendRange</code>	<code>window("Tool Panel").modal = FALSE</code>
<code>colorRange</code>	<code>numParticles</code>
<code>direction</code>	<code>path</code>

---

distribution	pathStrength
drag	region
gravity	sizeRange (contains end and start)
lifetime	texture
loop (emitter)	tweenMode
maxSpeed	wind

## Planes

Use these terms to control properties of 3D planes:

length (3D)	width (3D)
lengthVertices	widthVertices

## Spheres

Use these terms to control properties of 3D spheres:

endAngle	resolution
radius	state (3D)

## Selecting models

Use these terms to enable individual models in a 3D cast member to be selected and respond to mouse clicks. This is also known as *picking*.

modelsUnderLoc	spriteSpaceToWorldSpace
modelsUnderRay	worldSpaceToSpriteSpace
modelUnderLoc	

## Shaders

Use these terms to work with shaders:

ambient	renderStyle
blend (3D)	shadowPercentage
blendConstant	shaderList
blendConstantList	shadowPercentage
blendFunction	shadowStrength
blendFunctionList	silhouettes
blendSource	specular (shader)
blendSourceList	specularColor
count	specularLightMap
deleteShader	style
diffuse	textureMode

<code>diffuseColor</code>	<code>textureModelList</code>
<code>diffuseLightMap</code>	<code>textureRepeat</code>
<code>emissive</code>	<code>textureRepeatList</code>
<code>flat</code>	<code>textureTransform</code>
<code>glossMap</code>	<code>textureTransformList</code>
<code>name</code>	<code>transparent</code>
<code>newShader</code>	<code>type (shader)</code>
<code>renderStyle</code>	<code>useDiffuseWithTexture</code>
<code>region</code>	<code>wrapTransformList</code>
<code>reflectivity</code>	

## Engraver shader

Use these terms to work with the Engraver shader:

<code>density</code>	<code>rotation (engraver shader)</code>
<code>brightness</code>	

## Newsprint shader

Use these terms to work with the Newsprint shader:

<code>density</code>	<code>brightness</code>
----------------------	-------------------------

## Painter shader

Use these terms to work with the Painter shader:

<code>colorSteps</code>	<code>shadowPercentage</code>
<code>highlightPercentage</code>	<code>shadowStrength</code>
<code>highlightStrength</code>	<code>style</code>

## Sprites (3D)

<code>rect (camera)</code>	<code>registerForEvent()</code>
----------------------------	---------------------------------

Use these terms to control properties of 3D sprites:

## Streaming

Use these terms to control the streaming of 3D cast members:

<code>bytesStreamed (3D)</code>	<code>state (3D)</code>
<code>preload (3D)</code>	<code>streamSize (3D)</code>

## Subdivision surfaces modifier

Use these terms to control the functionality of the subdivision surfaces (SDS) modifier:

depth (3D)	sds (modifier)
enabled (sds)	subdivision
error	tension

## Text (3D)

Use these terms to control the appearance of 3D text:

autoCameraPosition	displayMode
bevelDepth	extrude3D
bevelType	smoothness
displayFace	tunnelDepth

## Textures

Use these terms to work with textures:

compressed	newTexture
count	quality (3D)
deleteTexture	renderFormat
height (3D)	texture
member	textureRenderFormat
name	textureType
nearFiltering	type (texture)

## Toon modifier

Use these terms to control the functionality of the Toon modifier:

boundary	lineOffset
colorSteps	shadowPercentage
creaseAngle	shadowStrength
creases	silhouettes
highlightPercentage	style
highlightStrength	toon (modifier)
lineColor	useLineOffset

## Transforms

Use these terms to work with transforms:

---

<code>duplicate</code>	<code>preRotate</code>
<code>getWorldTransform()</code>	<code>preScale()</code>
<code>identity()</code>	<code>preTranslate()</code>
<code>interpolate()</code>	<code>rotate</code>
<code>interpolateTo()</code>	<code>rotation (transform)</code>
<code>inverse()</code>	<code>scale (transform)</code>
<code>invert()</code>	<code>transform (property)</code>
<code>multiply()</code>	<code>translate</code>
<code>pointAt</code>	<code>worldPosition</code>
<code>pointAtOrientation</code>	<code>xAxis</code>
<code>position (transform)</code>	<code>yAxis</code>
<code>preMultiply</code>	<code>zAxis</code>

---

## Vector math

Use these terms to perform vector math operations:

---

<code>angleBetween</code>	<code>getNormalized</code>
<code>axisAngle</code>	<code>magnitude</code>
<code>cross</code>	<code>normalize</code>
<code>crossProduct()</code>	<code>randomVector</code>
<code>distanceTo()</code>	<code>vector()</code>
<code>dot()</code>	<code>x (vector property)</code>
<code>dotProduct()</code>	<code>y (vector property)</code>
<code>duplicate</code>	<code>z (vector property)</code>

---

# CHAPTER 3

## Lingo Dictionary

This dictionary describes the syntax and use of Lingo elements in Macromedia Director MX. Nonalphabetical operators are presented first, followed by all other operators in alphabetical order.

The entries in this dictionary are the same as those in Director Help. To use examples in a script, copy the example text from Director Help and paste it in the Script window.

### # (symbol)

#### Syntax

*#symbolName*

#### Description

Symbol operator; defines a symbol, a self-contained unit that can be used to represent a condition or flag. The value *symbolName* begins with an alphabetical character and may be followed by any number of alphabetical or numerical characters.

A symbol can do the following:

- Assign a value to a variable.
- Compare strings, integers, rectangles, and points.
- Pass a parameter to a handler or method.
- Return a value from a handler or method.

A symbol takes up less space than a string and can be manipulated, but unlike a string it does not consist of individual characters. You can convert a symbol to a string for display purposes by using the `string` function.

The following are some important points about symbol syntax:

- Symbols are not case-sensitive.
- Symbols can't start with a number.
- Spaces may not be used, but you can use underscore characters to simulate them.
- Symbols use the 128 ASCII characters, and letters with diacritical or accent marks are treated as their base letter.
- Periods may not be used in symbols.

All symbols, global variables, and names of parameters passed to global variables are stored in a common lookup table.

#### Example

This statement sets the state variable to the symbol `#Playing`:

```
state = #Playing
```

#### See also

```
ilk(), string(), symbol(), symbolP()
```

## . (dot operator)

#### Syntax

```
objectReference.objectProperty  
textExpression.objectProperty  
object.commandOrFunction()
```

#### Description

Operator; used to test or set properties of objects, or to issue a command or execute a function of the object. The object may be a cast member, a sprite, a property list, a child object of a parent script, or a behavior.

#### Examples

This statement displays the current member contained by the sprite in channel 10:

```
put sprite(10).member
```

To use the alternate syntax and call a function, you can use this form:

```
myColorObject = color(#rgb, 124, 22, 233)  
put myColorObject.ilk()  
-- #color
```

## - (minus)

#### Syntax

```
(Negation): -expression
```

#### Description

Math operator; reverses the sign of the value of *expression*.

This is an arithmetic operator with a precedence level of 5.

#### Syntax

```
(Subtraction): expression1 - expression2
```

#### Description

Math operator; performs an arithmetic subtraction on two numerical expressions, subtracting *expression2* from *expression1*. When both expressions are integers, the difference is an integer. When either or both expressions are floating-point numbers, the difference is a floating-point number.

This is an arithmetic operator with a precedence level of 3.



### Examples

(Negation): This statement reverses the sign of the expression  $2 + 3$ :

```
put -(2 + 3)
```

The result is -5.

(Subtraction): This statement subtracts the integer 2 from the integer 5 and displays the result in the Message window:

```
put 5 - 2
```

The result is 3, which is an integer.

(Subtraction): This statement subtracts the floating-point number 1.5 from the floating-point number 3.25 and displays the result in the Message window:

```
put 3.25 - 1.5
```

The result is 1.75, which is a floating-point number.

## -- (comment)

### Syntax

```
-- comment
```

### Description

Comment delimiter; indicates the beginning of a script comment. On any line, anything that appears between the comment delimiter (double hyphen) and the end-of-line return character is interpreted as a comment rather than a Lingo statement.

The Director player for Java accepts Lingo that uses this delimiter, but comments do not appear in the final Java code.

### Example

This handler uses a double hyphen to make the second, fourth, and sixth lines comments:

```
on resetColors
  -- This handler resets the sprite's colors.
  sprite(1).forecolor = 35
  -- bright red
  sprite(1).backcolor = 36
  -- light blue
end
```

## & (concatenator)

### Syntax

```
expression1 & expression2
```

### Description

String operator; performs a string concatenation of two expressions. If either *expression1* or *expression2* is a number, it is first converted to a string. The resulting expression is a string.

This is a string operator with a precedence level of 2.

Be aware that Lingo allows you to use some commands and functions that take only one argument without parentheses surrounding the argument. When an argument phrase includes an operator, Lingo interprets only the first argument as part of the function, which may confuse Lingo.

For example, the `open window` command allows one argument that specifies which window to open. If you use the `&` operator to define a pathname and filename, Director interprets only the string before the `&` operator as the filename. For example, Lingo interprets the statement `open window the applicationPath & "theMovie"` as `(open window the applicationPath) & ("theMovie")`. Avoid this problem by placing parentheses around the entire phrase that includes an operator, as follows:

```
open window (the applicationPath & "theMovie")
```

The parentheses clear up Lingo's confusion by changing the precedence by which Lingo deals with the operator, causing Lingo to treat the two parts of the argument as one complete argument.

### Examples

This statement concatenates the strings "abra" and "cadabra" and displays the resulting string in the Message window:

```
put "abra" & "cadabra"
```

The result is the string "abracadabra".

This statement concatenates the strings "\$" and the content of the `price` variable and then assigns the concatenated string to the `Price` field cast member:

```
member("Price").text = "$" & price
```

## && (concatenator)

### Syntax

```
expression1 && expression2
```

### Description

String operator; concatenates two expressions, inserting a space character between the original string expressions. If either *expression1* or *expression2* is a number, it is first converted to a string. The resulting expression is a string.

This is a string operator with a precedence level of 2.

### Examples

This statement concatenates the strings "abra" and "cadabra" and inserts a space between the two:

```
put "abra" && "cadabra"
```

The result is the string "abra cadabra".

This statement concatenates the strings "Today is" and today's date in the long format and inserts a space between the two:

```
put "Today is" && the long date
```

If today's date is Tuesday, July 30, 2000, the result is the string "Today is Tuesday, July 30, 2000".

## () (parentheses)

### Syntax

*(expression)*

### Description

Grouping operator; performs a grouping operation on an expression to control the order of execution of the operators in an expression. This operator overrides the automatic precedence order so that the expression within the parentheses is evaluated first. When parentheses are nested, the contents of the inner parentheses are evaluated before the contents of the outer ones.

This is a grouping operator with a precedence level of 5.

Be aware that Lingo allows you to use some commands and functions that take only one argument without parentheses surrounding the argument. When an argument phrase includes an operator, Lingo interprets only the first argument as part of the function, which may confuse Lingo.

For example, the `open window` command allows one argument that specifies which window to open. If you use the `&` operator to define a pathname and filename, Director interprets only the string before the `&` operator as the filename. For example, Lingo interprets the statement `open window the applicationPath & "theMovie"` as `(open window the applicationPath) & ("theMovie")`. Avoid this problem by placing parentheses around the entire phrase that includes an operator, as follows:

```
open window (the applicationPath & "theMovie")
```

### Example

These statements use the grouping operator to change the order in which operations occur (the result appears below each statement):

```
put (2 + 3) * (4 + 5)
-- 45
put 2 + (3 * (4 + 5))
-- 29
put 2 + 3 * 4 + 5
-- 19
```

## \* (multiplication)

### Syntax

*expression1 \* expression2*

### Description

Math operator; performs an arithmetic multiplication on two numerical expressions. If both expressions are integers, the product is an integer. If either or both expressions are floating-point numbers, the product is a floating-point number.

This is an arithmetic operator with a precedence level of 4.

### Examples

This statement multiplies the integers 2 and 3 and displays the result in the Message window:

```
put 2 * 3
```

The result is 6, which is an integer.

This statement multiplies the floating-point numbers 2.0 and 3.1414 and displays the result in the Message window:

```
put 2.0 * 3.1416
```

The result is 6.2832, which is a floating-point number.

## + (addition)

### Syntax

*expression1 + expression2*

### Description

Math operator; performs an arithmetic sum on two numerical expressions. If both expressions are integers, the sum is an integer. If either or both expressions are floating-point numbers, the sum is a floating-point number.

This is an arithmetic operator with a precedence level of 4.

### Examples

This statement adds the integers 2 and 3 and then displays the result, 5, an integer, in the Message window:

```
put 2 + 3
```

This statement adds the floating-point numbers 2.5 and 3.25 and displays the result, 5.7500, a floating-point number, in the Message window:

```
put 2.5 + 3.25
```

## + (addition) (3D)

### Syntax

*vector1 + vector2*  
*vector + scalar*

### Description

3D vector operator; adds the components of two vectors, or adds the scalar value to each component of the vector and returns a new vector.

*vector1 + vector2* adds the components of *vector1* to the corresponding components of *vector2* and returns a new vector.

*vector + scalar* adds the scalar value to each of the components of the vector and returns a new vector.

## - (subtraction)

### Syntax

*vector1 - vector2*  
*vector - scalar*

### Description

3D vector operator; subtracts the components of *vector2* from the corresponding components of *vector1*, or subtracts the scalar value from each of the components and returns a new vector.

*vector1* - *vector2* subtracts the values of *vector2* from the corresponding components in *vector1* and returns a new vector.

*vector* - *scalar* subtracts the value of the scalar from each of the components in the vector and returns a new vector.

## \* (multiplication)

### Syntax

```
vector1 * vector2  
vector * scalar  
transform * vector
```

### Description

3D vector operator; multiplies the components of *vector1* by the corresponding components in *vector2*, and returns the dot product, or multiplies each of the components the vector by the scalar value and returns a new vector.

*vector1* \* *vector2* returns the dot product of the two vectors, which is not a new vector. This operation is the same as *vector1*.dotproduct.*vector2*.

*vector* \* *scalar* multiplies each of the components in the vector by the scalar value and returns a new vector.

*transform* \* *vector* multiplies the *transform* by the *vector* and returns a new vector. The new vector is the result of applying the positional and rotational changes defined by *transform* to the *vector*. Note that *vector* \* *transform* is not supported.

### See also

dotProduct()

## / (division)

### Syntax

```
expression1 / expression2
```

### Description

Math operator; performs an arithmetic division on two numerical expressions, dividing *expression1* by *expression2*. If both expressions are integers, the quotient is an integer. If either or both expressions are floating-point numbers, the quotient is a floating-point number.

This is an arithmetic operator with a precedence level of 4.

### Examples

This statement divides the integer 22 by 7 and then displays the result in the Message window:

```
put 22 / 7
```

The result is 3. Because both numbers in the division are integers, Lingo rounds the answer down to the nearest integer.

This statement divides the floating-point number 22.0 by 7.0 and then displays the result in the Message window:

```
put 22.0 / 7.0
```

The result is 3.1429, which is a floating-point number.

## / (division) (3D)

### Syntax

*vector / scalar*

### Description

3D vector operator; divides each of the vector components by the scalar value and returns a new vector.

## < (less than)

### Syntax

*expression1 < expression2*

### Description

Comparison operator; compares two expressions and determines whether *expression1* is less than *expression2* (TRUE), or whether *expression1* is greater than or equal to *expression2* (FALSE).

This operator can compare strings, integers, floating-point numbers, rects, and points. Be aware that comparisons performed on rects or points are handled as if the terms were lists, with each element of the first list compared to the corresponding element of the second list.

This is a comparison operator with a precedence level of 1.

## <= (less than or equal to)

### Syntax

*expression1 <= expression2*

### Description

Comparison operator; compares two expressions and determines whether *expression1* is less than or equal to *expression2* (TRUE), or whether *expression1* is greater than *expression2* (FALSE).

This operator can compare strings, integers, floating-point numbers, rects, and points. Be aware that comparisons performed on rects or points are handled as if the terms were lists, with each element of the first list compared to the corresponding element of the second list.

This is a comparison operator with a precedence level of 1.

## <> (not equal)

### Syntax

*expression1 <> expression2*

### Description

Comparison operator; compares two expressions, symbols, or operators and determines whether *expression1* is not equal to *expression2* (TRUE), or whether *expression1* is equal to *expression2* (FALSE).

This operator can compare strings, integers, floating-point numbers, rects, and points. Be aware that comparisons performed on rects or points are handled as if the terms were lists, with each element of the first list compared to the corresponding element of the second list.

This is a comparison operator with a precedence level of 1.

## = (equals)

### Syntax

*expression1* = *expression2*

### Description

Comparison operator; compares two expressions, symbols, or objects and determines whether *expression1* is equal to *expression2* (TRUE), or whether *expression1* is not equal to *expression2* (FALSE).

This operator can compare strings, integers, floating-point numbers, rects, lists, and points.

Lists are compared based on the number of elements in the list. The list with more elements is considered larger than the list with fewer elements.

This is a comparison operator with a precedence level of 1.

## > (greater than)

### Syntax

*expression1* > *expression2*

### Description

Comparison operator; compares two expressions and determines whether *expression1* is greater than *expression2* (TRUE), or whether *expression1* is less than or equal to *expression2* (FALSE).

This operator can compare strings, integers, floating-point numbers, rects, and points. Be aware that comparisons performed on rects or points are handled as if the terms were lists, with each element of the first list compared to the corresponding element of the second list.

This is a comparison operator with a precedence level of 1.

## >= (greater than or equal to)

### Syntax

*expression1* >= *expression2*

### Description

Comparison operator; compares two expressions and determines whether *expression1* is greater than or equal to *expression2* (TRUE), or whether *expression1* is less than *expression2* (FALSE).

This operator can compare strings, integers, floating-point numbers, rects, and points. Be aware that comparisons performed on rectangles or points are handled as if the terms were lists, with each element of the first list compared to the corresponding element of the second list.

This is a comparison operator with a precedence level of 1.

## [ ] (bracket access)

### Syntax

*textExpression*[*chunkNumberBeingAddressed*]  
*textExpression*[*firstChunk*..*lastChunk*]

### Description

Operator; allows a chunk expression to be addressed by number. Useful for finding the *n*th chunk in the expression. The chunk can be a word, line, character, paragraph, or other Text cast member chunk.

### Example

This outputs the first word of the third line in the text cast member First Names:

```
put member("First Names").text.line[3].word[1]
```

## [ ] (list)

### Syntax

[*entry1*, *entry2*, *entry3*, ...]

### Description

List operator; specifies that the entries within the brackets are one of four types of lists:

- Unsorted linear lists
- Sorted linear lists
- Unsorted property lists
- Sorted property lists

Each entry in a linear list is a single value that has no other property associated with it. Each entry in a property list consists of a property and a value. The property appears before the value and is separated from the value by a colon. You cannot store a property in a linear list. When using strings as entries in a list, enclose the string in quotation marks.

For example, [6, 3, 8] is a linear list. The numbers have no properties associated with them. However, [#gears:6, #balls:3, #ramps:8] is a property list. Each number has a property—in this case, a type of machinery—associated with it. This property list could be useful for tracking the number of each type of machinery currently on the Stage in a mechanical simulation. Properties can appear more than once in a property list.

Lists can be sorted in alphanumeric order. A sorted linear list is ordered by the values in the list. A sorted property list is ordered by the properties in the list. You sort a list by using the appropriate command for a linear list or property list.

- In linear lists, symbols and strings are case sensitive.
- In property lists, symbols aren't case sensitive, but strings are case sensitive.

A linear list or property list can contain no values at all. An empty list consists of two square brackets ([ ]). To create or clear a linear list, set the list to [ ]. To create or clear a property list, set the list to [:].

You can modify, test, or read items in a list.



Lingo treats an instance of a list as a reference to the list. This means each instance is the same piece of data, and changing it will change the original. Use the `duplicate` command to create copies of lists.

Lists are automatically disposed when they are no longer referred to by any variable. When a list is held within a global variable, it persists from movie to movie.

You can initialize a list in the `on prepareMovie` handler or write the list as a field cast member, assign the list to a variable, and then handle the list by handling the variable.

Not all PC keyboards have square brackets. If square brackets aren't available, use the `list` function to create a linear list.

For a property list, create the list pieces as a string before converting them into a useful list.

```
myListString = numToChar(91) & ":" & numToChar(93)
put myListString
-- "[" "]"
myList = myListString.value
put myList
-- [:]
put myList.listP
-- 1
myList[#name] = "Brynn"
put myList
-- [#name: "Brynn"]
```

### Examples

This statement defines a list by making the `machinery` variable equal to the list:

```
set machinery = [#gears:6, #balls:3, #ramps:8]
```

This handler sorts the list `aList` and then displays the result in the Message window:

```
on sortList aList
    alist.sort()
    put alist
end sortList
```

If the movie issues the statement `sortList machinery`, where `machinery` is the list in the preceding example, the result is `[#balls:3, #gears:6, #ramps:8]`.

This statement creates an empty linear list:

```
set x = [ ]
```

This statement creates an empty property list:

```
set x = [:]
```

### See also

`add`, `addVertex`, `append`, `count()`, `deleteAt`, `duplicate()` (list function), `findPos`, `findPosNear`, `getProp()`, `getAt`, `getLast()`, `getPos()`, `ilk()`, `list()`, `max()`, `min`, `setAt`, `setaProp`, `sort`

## " (string)

### Syntax

"

### Description

String constant; when used before and after a string, quotation marks indicate that the string is a literal—not a variable, numerical value, or Lingo element. Quotation marks must always surround literal names of cast members, casts, windows, and external files.

### Example

This statement uses quotation marks to indicate that the string “San Francisco” is a literal string, the name of a cast member:

```
put member("San Francisco").loaded
```

### See also

QUOTE

## ␣ (continuation)

### Description

The ␣ symbol is obsolete. Use the \ character instead. See \ (continuation). It is recommended that you replace this symbol with the \ symbol in your older scripts.

## \ (continuation)

### Syntax

*first part of a statement on this line \*  
*second part of the statement \*  
*third part of the statement*

### Description

Continuation symbol; when used as the last character in a line, indicates that the statement continues on the next line. Lingo then interprets the lines as one continuous statement.

### Example

This statement uses the \ character to wrap the statement onto two lines:

```
set the memberNum of sprite mySprite \  
to member "This is a long cast name."
```

## @ (pathname)

### Syntax

@*pathReference*

### Description

Pathname operator; defines the path to the current movie's folder and is valid on both Windows and Macintosh computers.

Identify the current movie's folder by using the @ symbol followed by one of these pathname separators:

- / (forward slash)
- \ (backslash)
- : (colon)

When a movie is queried to determine its location, the string returned will include the @ symbol.

Be sure to use only the @ symbol when navigating between Director movies or changing the source of a linked media cast member. The @ symbol does not work when the FileIO Xtra or other functions are used outside those available within Director.

You can build on this pathname to specify folders that are one or more levels above or below the current movie's folder. Keep in mind that the @ portion represents the current movie's location, not necessarily the location of the projector.

- Add an additional pathname separator immediately after the @ symbol to specify a folder one level up in the hierarchy.
- Add folder names and filenames (separated by /, \, or :) after the current folder name to specify subfolders and files within folders.

You can use relative pathnames in Lingo to indicate the location of a linked file in a folder different than the movie's folder.

### Examples

These are equivalent expressions that specify the subfolder bigFolder, which is in the current movie's folder:

```
@/bigFolder  
@:bigFolder  
@\bigFolder
```

These are equivalent expressions that specify the file linkedFile, in the subfolder bigFolder, which is in the current movie's folder:

```
@:bigFolder:linkedFile  
@\bigFolder\linkedFile  
@/bigFolder/linkedFile
```

### Examples

This expression specifies the file linkedFile, which is located one level up from the current movie's folder:

```
@//linkedFile
```

This expression specifies the file linkedFile, which is located two levels up from the current movie's folder:

```
@:::linkedFile
```

These are equivalent expressions that specify the file linkedFile, which is in the folder otherFolder. The otherFolder folder is in the folder one level up from the current movie's folder.

```
@::otherFolder:linkedFile  
@\otherFolder\linkedFile  
@//otherFolder/linkedFile
```

### See also

searchPath, fileName (cast property), fileName (cast member property), fileName (window property)

## abbr, abbrev, abbreviated

These elements are used by the `date` and `time` functions.

### See also

`date()` (`system clock`)

## abort

### Syntax

`abort`

### Description

Command; tells Lingo to exit the current handler and any handler that called it without executing any of the remaining statements in the handler. This differs from the `exit` keyword, which returns to the handler from which the current handler was called.

The `abort` command does not quit Director.

### Example

This statement instructs Lingo to exit the handler and any handler that called it when the amount of free memory is less than 50K:

```
if the freeBytes < 50*1024 then abort
```

### See also

`exit`, `halt`, `quit`

## abs()

### Syntax

`abs (numericExpression)`

### Description

Math function; calculates the absolute value of a numerical expression. If *numericExpression* is an integer, its absolute value is also an integer. If *numericExpression* is a floating-point number, its absolute value is also a floating-point number.

The `abs` function has several uses. It can simplify the tracking of mouse and sprite movement by converting coordinate differences (which can be either positive or negative numbers) into distances (which are always positive numbers). The `abs` function is also useful for handling mathematical functions, such as `sqrt` and `log`.

### Example

This statement determines whether the absolute value of the difference between the current mouse position and the value of the variable `startV` is greater than 30 (since you wouldn't want to use a negative number for distance). If it is, the foreground color of sprite 6 is changed.

```
if (the mouseV - startV).abs > 30 then sprite(6).forecolor = 95
```

## actionsEnabled

### Syntax

the actionsEnabled of sprite *whichFlashSprite*  
the actionsEnabled of member *whichFlashMember*  
sprite *whichFlashSprite*.actionsEnabled  
member *whichFlashMember*.actionsEnabled

### Description

Cast member property and sprite property; controls whether the actions in a Flash movie are enabled (TRUE, default) or disabled (FALSE).

This property can be tested and set.

### Example

This handler accepts a sprite reference as a parameter, and then toggles the sprite's actionsEnabled property on or off.

#### Dot syntax:

```
on ToggleActions whichSprite
    sprite (whichSprite).actionsEnabled = not sprite
    (whichSprite).actionsEnabled
end
```

#### Verbose syntax:

```
on ToggleActions whichSprite
    set the actionsEnabled of sprite whichSprite = not the actionsEnabled of
    sprite whichSprite
end
```

## activateApplication

### Syntax

```
on activateApplication
```

### Description

Built-in handler; runs when the projector is brought to the foreground. This handler is useful when a projector runs in a window and the user can send it to the background to work with other applications. When the projector is brought back to the foreground, this handler runs. Any MIAWs running in the projector can also make use of this handler.

During authoring, this handler is called only if Animate in Background is turned on in General Preferences.

On Windows, this handler is not called if the projector is merely minimized and no other application is brought to the foreground.

### Example

This handler plays a sound each time the user brings the projector back to the foreground:

```
on activateApplication
    sound(1).queue(member("openSound"))
    sound(1).play()
end
```

### See also

deactivateApplication, activeCastLib, on deactivateWindow

## on activateWindow

### Syntax

```
on activateWindow
    statement(s)
end
```

### Description

System message and event handler; contains statements that run in a movie when the user clicks the inactive window and the window comes to the foreground.

You can use an `on activateWindow` handler in a script that you want executed every time the movie becomes active.

Clicking the main movie (the main Stage) does not generate an `on activateWindow` handler.

### Example

This handler plays the sound Hurray when the window that the movie is playing in becomes active:

```
on activateWindow
    puppetSound 2, "Hurray"
end
```

### See also

`activateWindow`, `close window`, `on deactivateWindow`, `frontWindow`, `on moveWindow`, `open`

## active3dRenderer

### Syntax

```
the active3dRenderer
```

### Description

3D Lingo movie property; indicates the renderer currently in use by the movie for drawing 3D sprites. This property is equivalent to the `getRendererServices().renderer` property.

The possible values of the `active3dRenderer` property are `#openGL`, `#directX7_0`, `#directX5_2`, and `#software`. The values `#openGL`, `#directX7_0`, and `#directX5_2`, which are video card drivers, will lead to much faster performance than `#software`, a software renderer used when none of the first three options are available.

The `active3dRenderer` property can be tested, but not set. Use `getRendererServices().renderer` to set this property.

### Examples

These examples show the two ways to determine which renderer is currently in use.

```
put the active3dRenderer
-- #openGL
put getRendererServices().renderer
-- #openGL
```

### See also

`renderer`, `rendererDeviceList`, `getRendererServices()`

## activeCastLib

### Syntax

the activeCastLib

### Description

System property; indicates which cast was most recently activated. The activeCastLib property's value is the cast's number.

The activeCastLib property is useful when working with the selection castLib property. Use it to determine which cast the selection refers to.

This property can be tested but not set.

### Example

These statements assign the selected cast members in the most recently selected cast to the variable selectedMembers:

```
castLibOfInterest = the activeCastLib
selectedMembers = castLib(castLibOfInterest).selection
```

An equivalent way to write this is:

```
selectedMembers = castLib(the activeCastLib).selection
```

## activeWindow

### Syntax

the activeWindow

### Description

Movie property; indicates which movie window is currently active. For the main movie, activeWindow is the Stage. For a movie in a window, activeWindow is the movie in the window.

### Example

This example places the word Active in the title bar of the clicked window and places the word Inactive in the title bar of all other open windows:

```
on activateWindow
  set clickedWindow = (the windowlist).getPos(the activeWindow)
  set windowCount = (the windowlist).count
  repeat with x = 1 to windowCount
    if x = clickedWindow then
      (the activeWindow).title = "Active"
    else
      (the windowlist[x]).title = "Inactive"
    end if
  end repeat
end
```

### See also

activeCastLib, windowList

## actorList

### Syntax

the actorList

### Description

Movie property; a list of child objects that have been explicitly added to this list. Objects in `actorList` receive a `stepFrame` message each time the playhead enters a frame.

To add an object to the `actorList`, use `add actorList, theObject`. The object's `on stepFrame` handler in its parent or ancestor script will then be called automatically at each frame advance.

To clear objects from the `actorList`, set `actorList` to `[ ]`, which is an empty list.

Director doesn't clear the contents of `actorList` when branching to another movie, which can cause unpredictable behavior in the new movie. To prevent child objects in the current movie from being carried over to the new movie, insert the statement `set the actorList = [ ]` in the `on prepareMovie` handler of the new movie.

Unlike previous versions of Director, `actorList` is now supported in the Director player for Java.

### Examples

This statement adds a child object created from the parent script `Moving Ball`. All three values are parameters that the script requires.

```
add the actorList, new(script "MovingBall", 1, 200,200)
```

This statement displays the contents of `actorList` in the Message window:

```
put the actorList
```

This statement clears objects from `actorList`.

```
the actorList = [ ]
```

### See also

`new()`

## add

### Syntax

```
linearList.add(value)  
add linearList, value
```

### Description

List command; for linear lists only, adds the value specified by *value* to the linear list specified by *linearList*. For a sorted list, the value is placed in its proper order. For an unsorted list, the value is added to the end of the list.

This command returns an error when used on a property list.

**Note:** Don't confuse the `add` command with the `+` operator used for addition or the `&` operator used to concatenate strings.

### Examples

These statements add the value 2 to the list named `bids`. The resulting list is `[3, 4, 1, 2]`.

```
bids = [3, 4, 1]  
bids.add(2)
```

This statement adds 2 to the sorted linear list `[1, 4, 5]`. The new item remains in alphanumeric order because the list is sorted.

```
bids.add(2)
```



#### See also

`sort`

## add (3D texture)

#### Syntax

```
member(whichCastmember).model(whichModel).meshdeform.mesh[index].\
    textureLayer.add()
```

#### Description

3D `meshdeform` modifier command; adds an empty texture layer to the model's mesh.

You can copy texture coordinates between layers using the following code:

```
modelReference.meshdeform.texturelayer[a].texturecoordinatelist =
    modelReference.meshdeform.texturelayer[b].texturecoordinatelist
```

#### Example

This statement creates a new texture layer for the first mesh of the model named Ear.

```
member("Scene").model("Ear").meshdeform.mesh[1].\
    textureLayer.add()
```

#### See also

`meshDeform (modifier)`, `textureLayer`, `textureCoordinateList`

## addAt

#### Syntax

```
list.AddAt(position, value)
addAt list, position, value
```

#### Description

List command; for linear lists only, adds a value specified by *value* to a list at the position specified by *position*.

This command returns an error when used with a property list.

#### Example

This statement adds the value 8 to the fourth position in the list named `bids`, which is [3, 2, 4, 5, 6, 7]:

```
bids = [3, 2, 4, 5, 6, 7]
bids.addAt(4,8)
```

The resulting value of `bids` is [3, 2, 4, 8, 5, 6, 7].

## addBackdrop

#### Syntax

```
sprite(whichSprite).camera{( index )}.addBackdrop(texture, locWithinSprite,
    rotation)
member(whichCastmember).camera(whichCamera).addBackdrop(texture,
    locWithinSprite, rotation)
```

### Description

3D camera command; adds a backdrop to the end of the camera's list of backdrops. The backdrop is displayed in the 3D sprite at *locWithinSprite* with the indicated rotation. The *locWithinSprite* parameter is a 2D loc measured from the upper left corner of the sprite.

### Examples

The first line of this statement creates a texture named Rough from the cast member named Cedar and stores it in the variable t1. The second line applies the texture as a backdrop at the point (220, 220) within sprite 5. The texture has a rotation of 0 degrees. The last line applies the same texture as a backdrop for camera 1 of the cast member named Scene at the point (20, 20) with a rotation of 45 degrees.

```
t1 = member("Scene").newTexture("Rough", #fromCastMember, \
    member("Cedar"))
sprite(5).camera.addBackdrop(t1, point(220, 220), 0)
member("Scene").camera[1].addBackdrop(t1, point(20, 20), 45)
```

### See also

removeBackdrop

## addCamera

### Syntax

```
sprite(whichSprite).addCamera(whichCamera, index)
```

### Description

3D command; adds the camera *whichCamera*, at the given *index* position, to the list of cameras for the sprite. If *index* is greater than the value of *cameraCount()*, the camera is added to the end of the list. The view from each camera is displayed on top of the view from cameras with lower *index* positions. You can set the *rect* property of each camera to display multiple views within the sprite.

### Example

This statement inserts the camera named FlightCam at the fifth index position of the list of cameras of sprite 12:

```
sprite(12).addCamera(member("scene").camera("FlightCam"), 5)
```

### See also

cameraCount(), deleteCamera

# addChild

## Syntax

```
member(whichCastmember).node(whichParentNode).addChild(member\  
(whichCastmember).node(whichChildNode) {, #preserveWorld})
```

## Description

3D command; adds the node *whichChildNode* to the list of children of the node *whichParentNode*, and removes it from the list of children of its former parent. Either *node* argument can be a model, group, camera, or light. An equivalent operation would be to set the parent property of *whichChildNode* to *whichParentNode*.

The optional *#preserveWorld* parameter has two possible values: *#preserveWorld* or *#preserveParent*. When the child is added with *#preserveParent* specified, the parent-relative transform of the child remains unchanged and the child jumps to that transform in the space of its new parent. The child's world transform is recalculated. When the child is added with *#preserveWorld* specified, the world transform of the child remains unchanged and the child does not jump to its transform in the space of its new parent. Its parent-relative transform is recalculated.

## Examples

This statement adds the model named Tire to the list of children of the model named Car.

```
member("3D").model("Car").addChild(member("3D").model("Tire"))
```

This statement adds the model named Bird to the list of children of the camera named MyCamera and uses the *#preserveWorld* argument to maintain Bird's world position.

```
member("3D").camera("MyCamera").addChild(member("3D").model  
("Bird"), #preserveWorld)
```

## See also

parent, addToWorld, removeFromWorld

# addModifier

## Syntax

```
member(whichCastmember).model(whichModel).addModifier\  
(#modifierType)
```

## Description

3D model command; adds the specified modifier to the model. Possible modifiers are as follows:

- *#bonesPlayer*
- *#collision*
- *#inker*
- *#keyframePlayer*
- *#lod* (level of detail)
- *#meshDeform*
- *#sds*
- *#toon*

There is no default value for this command.

For more detailed information about each modifier, see the individual modifier entries.

### Example

This statement adds the toon modifier to the model named Box.

```
member("shapes").model("Box").addModifier(#toon)
```

### See also

bonesPlayer (modifier), collision (modifier), inker (modifier), keyframePlayer (modifier), lod (modifier), meshDeform (modifier), sds (modifier), toon (modifier), getRendererServices(), removeModifier, modifier, modifier[], modifiers

## addOverlay

### Syntax

```
sprite(whichSprite).camera{(index)}.addOverlay(texture, \
    locWithinSprite, rotation)
member(whichCastmember).camera(whichCamera).addOverlay(texture, \
    locWithinSprite, rotation)
```

### Description

3D camera command; adds an overlay to the end of the camera's list of overlays. The overlay is displayed in the 3D sprite at *locWithinSprite* with the indicated rotation. The *locWithinSprite* parameter is a 2D loc measured from the upper left corner of the sprite.

### Examples

The first line of this statement creates a texture named Rough from the cast member named Cedar and stores it in the variable t1. The second line applies the texture as an overlay at the point (220, 220) within sprite 5. The texture has a rotation of 0 degrees. The last line of the statement applies the same texture as an overlay for camera 1 of the cast member named Scene at the point (20, 20). The texture has a rotation of 45 degrees.

```
t1 = member("Scene").newTexture("Rough", #fromCastMember, \
    member("Cedar"))
sprite(5).camera.addOverlay(t1, point(220, 220), 0)
member("Scene").camera[1].addOverlay(t1, point(20, 20), 45)
```

### See also

removeOverlay

## addProp

### Syntax

```
list.addProp(property, value)
addProp list, property, value
```

### Description

Property list command; for property lists only, adds the property specified by *property* and its value specified by *value* to the property list specified by *list*. For an unsorted list, the value is added to the end of the list. For a sorted list, the value is placed in its proper order.

If the property already exists in the list, Lingo creates a duplicate property. You can avoid duplicate properties by using the setaProp command to change the new entry's property.

This command returns an error when used with a linear list.

### Examples

This statement adds the property named `kayne` and its assigned value 3 to the property list named `bids`, which contains [`#gee: 4, #ohasi: 1`]. Because the list is sorted, the new entry is placed in alphabetical order:

```
bids.addProp(#kayne, 3)
```

The result is the list [`#gee: 4, #kayne: 3, #ohasi: 1`].

This statement adds the entry `kayne: 7` to the list named `bids`, which now contains [`#gee: 4, #kayne: 3, #ohasi: 1`]. Because the list already contains the property `kayne`, Lingo creates a duplicate property:

```
bids.addProp(#kayne, 7)
```

The result is the list [`#gee: 4, #kayne: 3, #kayne: 7, #ohasi: 1`].

## addToWorld

### Syntax

```
member(whichCastmember).model(whichModel).addToWorld()  
member(whichCastmember).group(whichGroup).addToWorld()  
member(whichCastmember).camera(whichCamera).addToWorld()  
member(whichCastmember).light(whichLight).addToWorld()
```

### Description

3D command; inserts the model, group, camera, or light into the 3D world of the cast member as a child of the group named `World`.

When a model, group, camera, or light is created or cloned, it is automatically added to the world. Use the `removeFromWorld` command to take a model, group, camera, or light out of the 3D world without deleting it. Use the `isInWorld()` command to test whether a model, group, camera, or light has been added or removed from the world.

### Example

This statement adds the model named `gbCyl` to the 3D world of the cast member named `Scene`.

```
member("Scene").model("gbCyl").addToWorld()
```

### See also

`isInWorld()`, `removeFromWorld`

## addVertex

### Syntax

```
member(memberRef).AddVertex(indexToAddAt, pointToAddVertex \  
{, [ controlLocH, controlLocV ], [ controlLocH, controlLocV ]})  
addVertex(member memberRef, indexToAddAt, pointToAddVertex \  
{, [ controlLocH, controlLocV ], [ controlLocH, controlLocV ]})
```

### Description

Vector shape command; adds a new vertex to a vector shape cast member in the position specified.

The horizontal and vertical positions are relative to the origin of the vertex shape cast member.

When using the final two optional parameters, you can specify the location of the control handles for the vertex. The control handle location is offset relative to the vertex, so if no location is specified, it will be located at 0 horizontal offset and 0 vertical offset.

### Example

This line adds a vertex point in the vector shape Archie between the two existing vertex points, at the position 25 horizontal and 15 vertical:

```
member("Archie").addVertex(2, point(25, 15))
```

### See also

`vertexList`, `moveVertex()`, `deleteVertex()`, `originMode`

## after

### See

`put...after`

## alert

### Syntax

```
alert message
```

### Description

Command; causes a system beep and displays an alert dialog box containing the string specified by *message* and an OK button. This command is useful for providing error messages of up to 255 characters in your movie.

The message must be a string. If you want to include a number variable in an alert, use the `string` function to convert the variable to a string.

### Examples

The following statement produces an alert stating that there is no CD-ROM drive connected:

```
alert "There is no CD-ROM drive connected."
```

This statement produces an alert stating that a file was not found:

```
alert "The file" && QUOTE & filename & QUOTE && "was not found."
```

### See also

`string()`, `alertHook`

## alertHook

### Syntax

```
the alertHook
```

### Description

System property; specifies a parent script that contains the `on alertHook` handler. Use `alertHook` to control the display of alerts about file errors or Lingo script errors. When an error occurs and a parent script is assigned to `alertHook`, Director runs the `on alertHook` handler in the parent script.

Although it is possible to place `on alertHook` handlers in movie scripts, it is strongly recommended that you place an `on alertHook` handler in a behavior or parent script to avoid unintentionally calling the handler from a wide variety of locations and creating confusion about where the error occurred.

Because the `on alertHook` handler runs when an error occurs, avoid using the `on alertHook` handler for Lingo that isn't involved in handling an error. For example, the `on alertHook` handler is a bad location for a `go to movie` statement.

The `on alertHook` handler is passed an instance argument, two string arguments that describe the error, and an optional argument specifying an additional event that invokes the handler.

The fourth argument can have 1 of these 4 values:

- `#alert` - causes the handler to be triggered by the `alert` command.
- `#movie` - causes the handler to be triggered by a file not found error while performing a `go to movie` command.
- `#script` - causes the handler to be triggered by a script error.
- `#safeplayer` - causes the handler to be triggered by a check of the `safePlayer` property.

Depending on the Lingo within it, the `on alertHook` handler can ignore the error or report it in another way.

### Example

The following statement specifies that the parent script `Alert` is the script that determines whether to display alerts when an error occurs. If an error occurs, Lingo assigns the error and message strings to the field cast member `Output` and returns the value 1.

```
on prepareMovie
    the alertHook = script "Alert"
end

-- parent script "Alert"
on alertHook me, err, msg
    member("Output").text = err && msg
    return 1
end
```

### See also

`safePlayer`

## alignment

### Syntax

`member(whichCastMember).alignment`  
the alignment of member *whichCastMember*

### Description

Cast member property; determines the alignment used to display characters within the specified cast member. This property appears only to field and text cast members containing characters, if only a space.

For field cast members, the value of the property is a string consisting of one of the following: `left`, `center`, or `right`.

For text cast members, the value of the property is a symbol consisting of one of the following: `#left`, `#center`, `#right`, or `#full`.

The parameter *whichCastMember* can be either a cast name or a cast number.

This property can be tested and set. For text cast members, the property can be set on a per-paragraph basis.

### Examples

This statement sets the variable named `characterAlign` to the current alignment setting for the field cast member `Rokujo Speaks`:

Dot syntax:

```
characterAlign = member("Rokujo Speaks").alignment
```



**Verbose syntax:**

set characterAlign = the alignment of member "Rokujo Speaks"

This repeat loop consecutively sets the alignment of the field cast member Rove to left, center, and then right.

**Dot syntax:**

```
repeat with i = 1 to 3
    member("Rove").alignment = ("left center right").word[i]
end repeat
```

**Verbose syntax:**

```
repeat with i = 1 to 3
    set the alignment of member "Rove" to word i of "left center right"
end repeat
```

**See also**

text, font, lineHeight (cast member property), fontSize, fontStyle, & (concatenator), && (concatenator)

## allowCustomCaching

**Syntax**

the allowCustomCaching

**Description**

Movie property; will contain information regarding a private cache in future versions of Director.

This property defaults to TRUE, and can be tested and set.

**See also**

allowGraphicMenu, allowSaveLocal, allowTransportControl, allowVolumeControl, allowZooming

## allowGraphicMenu

**Syntax**

the allowGraphicMenu

**Description**

Movie property; sets the availability of the graphic controls in the context menu when playing the movie in a Shockwave environment.

Set this property to FALSE if you would rather have a text menu displayed than the graphic context menu.

This property defaults to TRUE, and can be tested and set.

**Example**

```
the allowGraphicMenu = 0
```

**See also**

allowSaveLocal, allowTransportControl, allowVolumeControl, allowZooming

## allowSaveLocal

### Syntax

the allowSaveLocal

### Description

Movie property; sets the availability of the Save control in the context menu when playing the movie in a Shockwave environment.

This property is provided to allow for enhancements in future versions of Shockwave.

This property defaults to TRUE, and can be tested and set.

### See also

allowGraphicMenu, allowTransportControl, allowVolumeControl, allowZooming

## allowTransportControl

### Syntax

the allowTransportControl

### Description

Movie property; This property is provided to allow for enhancements in future versions of Shockwave.

This property defaults to TRUE, and can be tested and set.

### See also

allowGraphicMenu, allowSaveLocal, allowVolumeControl, allowZooming

## allowVolumeControl

### Syntax

the allowVolumeControl

### Description

Movie property; sets the availability of the volume control in the context menu when playing the movie in a Shockwave environment.

When set to TRUE one or the other volume control is active, and is disabled when the property is set to FALSE.

This property defaults to TRUE, and can be tested and set.

### See also

allowGraphicMenu, allowSaveLocal, allowTransportControl, allowZooming

## allowZooming

### Syntax

the allowZooming

### Description

Movie property; determines whether the movie may be stretched or zoomed by the user when playing back in Shockwave and ShockMachine. Defaults to `TRUE`. This property can be tested and set. Set this property to `FALSE` to prevent users from changing the size of the movie in browsers and ShockMachine.

### See also

allowGraphicMenu, allowSaveLocal, allowTransportControl, allowVolumeControl

## alphaThreshold

### Syntax

member(*whichMember*).alphaThreshold  
the alphaThreshold of member *whichMember*

### Description

Bitmap cast member property; governs how the bitmap's alpha channel affects hit detection. This property is a value from 0 to 255, that exactly matches alpha values in the alpha channel for a 32-bit bitmap image.

For a given alphaThreshold setting, Director detects a mouse click if the pixel value of the alpha map at that point is equal to or greater than the threshold. Setting the alphaThreshold to 0 makes all pixels opaque to hit detection regardless of the contents of the alpha channel.

### See also

useAlpha

## ambient

### Syntax

member(*whichCastmember*).shader(*whichShader*).ambient  
member(*whichCastmember*).model(*whichModel*).shader.ambient  
member(*whichCastmember*).model(*whichModel*).shaderList[[*index*]].\ambient

### Description

3D #standard shader property; indicates how much of each color component of the ambient light in the cast member is reflected by the shader.

For example, if the color of the ambient light is `rgb(255, 255, 255)` and the value of the ambient property of the shader is `rgb(255, 0, 0)`, the shader will reflect all of the red component of the light that the shader's colors can reflect. However, it will reflect none of the blue and green components of the light, regardless of the colors of the shader. In this case, if there are no other lights in the scene, the blue and green colors of the shader will reflect no light, and will appear black.

The default value of this property is `rgb(63,63,63)`.

**Example**

This statement sets the `ambient` property of the model named `Chair` to `rgb(255, 255, 0)`. `Chair` will fully reflect the red and green components of the ambient light in the scene and completely ignore its blue component.

```
member("Room").model("Chair").shader.ambient = rgb(255, 0, 0)
```

**See also**

`ambientColor`, `newLight`, `type (light)`, `diffuse`, `specular (shader)`

## ambientColor

**Syntax**

```
member(whichCastmember).ambientColor
```

**Description**

3D cast member property; indicates the RGB color of the default ambient light of the cast member.

The default value for this property is `rgb(0, 0, 0)`. This adds no light to the scene.

**Example**

This statement sets the `ambientColor` property of the cast member named `Room` to `rgb(255, 0, 0)`. The default ambient light of the cast member will be red. This property can also be set in the Property inspector.

```
member("Room").ambientColor = rgb(255, 0, 0)
```

**See also**

`directionalColor`, `directionalPreset`, `ambient`

## ancestor

**Syntax**

```
property {optionalProperties} ancestor
```

**Description**

Object property; allows child objects and behaviors to use handlers that are not contained within the parent script or behavior.

The `ancestor` property is typically used with two or more parent scripts. You can use this property when you want child objects and behaviors to share certain behaviors that are inherited from an ancestor, while differing in other behaviors that are inherited from the parents.

For child objects, the `ancestor` property is usually assigned in the `on new` handler within the parent script. Sending a message to a child object that does not have a defined handler forwards that message to the script defined by the `ancestor` property.

If a behavior has an ancestor, the ancestor receives mouse events such as `mouseDown` and `mouseWithin`.

The `ancestor` property lets you change behaviors and properties for a large group of objects with a single command.

The ancestor script can contain independent property variables that can be obtained by child objects. To refer to property variables within the ancestor script, you must use this syntax:

```
me.propertyVariable = value
```

For example, this statement changes the property variable `legCount` within an ancestor script to 4:

```
me.legCount = 4
```

Use the syntax the *variableName* of *scriptName* to access property variables that are not contained within the current object. This statement allows the variable `myLegCount` within the child object to access the property variable `legCount` within the ancestor script:

```
set myLegCount to the legCount of me
```

### Example

Each of the following scripts is a cast member. The ancestor script `Animal` and the parent scripts `Dog` and `Man` interact with one another to define objects.

The first script, `Dog`, sets the property variable `breed` to `Mutt`, sets the ancestor of `Dog` to the `Animal` script, and sets the `legCount` variable that is stored in the ancestor script to 4:

```
property breed, ancestor
on new me
    set breed = "Mutt"
    set the ancestor of me to new(script "Animal")
    set the legCount of me to 4
    return me
end
```

The second script, `Man`, sets the property variable `race` to `Caucasian`, sets the ancestor of `Man` to the `Animal` script, and sets the `legCount` variable that is stored in the ancestor script to 2:

```
property race, ancestor
on new me
    set race to "Caucasian"
    set the ancestor of me to new(script "Animal")
    set the legCount of me to 2
    return me
end
```

### See also

`new()`, `me`, `property`

## and

### Syntax

```
logicalExpression1 and logicalExpression2
```

### Description

Logical operator; determines whether both *logicalExpression1* and *logicalExpression2* are TRUE (1), or whether either or both expressions are FALSE (0).

The precedence level of this logical operator is 4.

### Examples

This statement determines whether both logical expressions are TRUE and displays the result in the Message window:

```
put 1 < 2 and 2 < 3
```

The result is 1, which is the numerical equivalent of `TRUE`.

The first logical expression in the following statement is `TRUE`; and the second logical expression is `FALSE`. Because both logical expressions are not `TRUE`, the logical operator displays the result 0, which is the numerical equivalent of `FALSE`.

```
put 1 < 2 and 2 < 1
-- 0
```

**See also**

`not`, `or`

## angle

**Syntax**

```
member(whichCastmember).modelResource(whichModelResource).\
    emitter.angle
```

**Description**

3D emitter property; describes the area into which the particles of a particle system are emitted. A particle system is a model resource whose type is `#particle`.

The primary direction of particle emission is the vector set by the emitter's `direction` property. However, the direction of emission of a given particle will deviate from that vector by a random angle between 0 and the value of the emitter's `angle` property.

The effective range of this property is 0.0 to 180.0. The default value is 180.0.

**Example**

This statement sets the angle of emission of the model resource named `mrFount` to 1, which causes the emitted particles to form a thin line.

```
member("fountain").modelResource("mrFount").emitter.angle = 1
```

**See also**

`emitter`, `direction`

## angleBetween

**Syntax**

```
vector1.angleBetween(vector2)
```

**Description**

3D vector method; returns the angle between two vectors, in degrees.

**Example**

In this example, `pos1` is a vector on the X axis and `pos2` is a vector on the Y axis. The angle between these two vectors is 90°. The value returned by `pos1.angleBetween(pos2)` is 90.0000.

```
pos1 = vector(100, 0, 0)
pos2 = vector(0, 100, 0)
put pos1.angleBetween(pos2)
-- 90.0000
```

**See also**

`dot()`, `dotProduct()`

## animationEnabled

### Syntax

`member(whichCastmember).animationEnabled`

### Description

3D cast member property; indicates whether motions will be executed (TRUE) or ignored (FALSE). This property can also be set in the Property inspector.

The default value for this property is TRUE.

### Example

This statement disables animation for the cast member named Scene.

```
member("Scene").animationEnabled = FALSE
```

## antiAlias

### Syntax

`member(whichMember).antiAlias`  
`sprite(whichVectorSprite).antiAlias`

### Description

Cast member property; controls whether a text, Vector shape, or Flash cast member is rendered using anti-aliasing to produce high-quality rendering, but possibly slower playback of the movie. The `antiAlias` property is TRUE by default.

For vector shapes, TRUE is the equivalent of the #high quality setting for a Flash asset, and FALSE is the equivalent of #low.

The `antiAlias` property may also be used as a sprite property only for Vector shape sprites.

This property can be tested and set.

### Example

This behavior checks the color depth of the computer on which the movie is playing. If the color depth is set to 8 bits or less (256 colors), the script sets the `antiAlias` property of the sprite to FALSE.

```
property spriteNum
on beginsprite me
  if the colorDepth <= 8 then
    sprite(spriteNum).antiAlias = FALSE
  end if
end
```

### See also

`antiAliasThreshold`, `quality`

## antiAliasingEnabled

### Syntax

`sprite(whichSprite).antiAliasingEnabled`

### Type

3D sprite property.

### Description

This property indicates whether the 3D world in the sprite *whichSprite* is anti-aliased. It can be tested and set. The default value is `FALSE`, indicating that anti-aliasing is off. If the `antiAliasingEnabled` property is set to `TRUE` and the 3D renderer changes to a renderer that does not support anti-aliasing, the property is set to `FALSE`. The value of this property is not saved when the movie is saved.

Anti-aliased sprites use more processor power and memory than sprites that are not anti-aliased. Temporarily turning off anti-aliasing can improve the performance of animations and user interaction.

### Example

This Lingo checks whether the currently running 3D renderer for sprite 2 supports anti-aliasing with the `antiAliasingSupported` property. If anti-aliasing is supported, the second statement turns on anti-aliasing for the sprite with the `antiAliasingEnabled` property.

```
if sprite(2).antiAliasingSupported = TRUE then
    sprite(2).antiAliasingEnabled = TRUE
end if
```

### See Also

`antiAliasingSupported`, `renderer`, `rendererDeviceList`

## antiAliasingSupported

### Syntax

`sprite(whichSprite).antiAliasingSupported`

### Type

3D sprite property.

### Description

This property indicates whether anti-aliasing is supported by the current 3D renderer. This property can be tested but not set. This property returns either `TRUE` or `FALSE`.

### Example

This Lingo checks whether the currently running 3D renderer for sprite 3 supports anti-aliasing. If anti-aliasing is supported, the second statement turns on anti-aliasing for the sprite with the `antiAliasingEnabled` property.

```
if sprite(3).antiAliasingSupported = TRUE then
    sprite(3).antiAliasingEnabled = TRUE
end if
```

### See Also

`antiAliasingEnabled`, `renderer`, `rendererDeviceList`



## antiAliasThreshold

### Syntax

`member(whichTextMember).antiAliasThreshold`

### Description

Text cast member property; this setting controls the point size at which automatic anti-aliasing takes place in a text cast member. This has an effect only when the `antiAlias` property of the text cast member is set to `TRUE`.

The setting itself is an integer indicating the font point size at which the anti-alias takes place.

This property defaults to 14 points.

### See also

`antiAlias`

## append

### Syntax

```
list.append(value)  
append list, value
```

### Description

List command; for linear lists only, adds the specified value to the end of a linear list. This differs from the `add` command, which adds a value to a sorted list according to the list's order.

This command returns a script error when used with a property list.

### Example

This statement adds the value 2 at the end of the sorted list named `bids`, which contains `[1, 3, 4]`, even though this placement does not match the list's sorted order:

```
set bids = [1, 3, 4]  
bids.append(2)
```

The resulting value of `bids` is `[1, 3, 4, 2]`.

### See also

`add (3D texture)`, `sort`

## applicationPath

### Syntax

`the applicationPath`

### Description

System property; determines the path or location of the folder containing the running copy of the Director application during authoring, or the folder containing the projector during run time. The property value is a string.

If you use the `applicationPath` followed by `&` and a path to a subfolder, enclose the entire expression in parentheses so that Lingo parses the expression as one phrase.

The Director player for Java doesn't support this property, nor does Shockwave.

This property can be tested but not set.

### Examples

This statement displays the pathname for the folder that contains the Director application.

```
put the applicationPath  
-- "Z:\Program Files\Macromedia\Director"
```

This statement opens the movie Sunset Boulevard in a window (on a Windows machine):

```
open window (the applicationPath & "\Film Noir\Sunset Boulevard")
```

### See also

@ (pathname), moviePath

## appMinimize

### Syntax

```
appMinimize
```

### Description

Command; on Windows, appMinimize causes a projector to minimize to the Windows Task Bar. On the Macintosh, appMinimize causes a projector to be hidden. Once hidden, the projector may be re-opened from the Macintosh application menu.

This is useful for projectors and MIAWs that play back without a title bar.

### See also

windowType

## atan()

### Syntax

```
(number).atan  
atan (number)
```

### Description

Math function; calculates the arctangent, which is the angle whose tangent is a specified number. The result is a value in radians between  $\pi/2$  and  $+\pi/2$ .

### Examples

This statement displays the arctangent of 1:

```
(1).atan
```

The result, to four decimal places, is 0.7854, or approximately  $\pi/4$ .

Note that most trigonometric functions use radians, so you may want to convert from degrees to radians.

This handler lets you convert between degrees and radians:

```
on DegreesToRads degreeValue  
    return degreeValue * PI/180  
end
```

The handler displays the conversion of 30 degrees to radians in the Message window:

```
put DegreesToRads(30)  
-- 0.5236
```

**See also**

`cos()`, `PI`, `sin()`

## attenuation

**Syntax**

```
member(whichCastMember).light(whichLight).attenuation
```

**Description**

3D light property; indicates the constant, linear, and quadratic attenuation factors for spotlights and point lights.

The default value for this property is `vector(1.0, 0.0, 0.0)`.

**Example**

This statement sets the `attenuation` property of the light named `HouseLight` to the vector `(.5, 0, 0)`, darkening it slightly.

```
member("3d world").light("HouseLight").attenuation = \
    vector(.5, 0, 0)
```

**See also**

`color (light)`

## attributeName

**Syntax**

```
XMLnode.attributeName[ attributeNumber ]
```

**Description**

XML property; returns the name of the specified child node of a parsed XML document.

**Example**

Beginning with the following XML:

```
<?xml version="1.0"?>
  <e1>
    <tagName attr1="val1" attr2="val2"/>
    <e2>element 2</e2>
    <e3>element 3</e3>
    here is some text
  </e1>
```

- This Lingo returns the name of the first attribute of the tag called `tagName`:

```
put gParserObject.child[1].child[1].attributeName[1]
-- "attr1"
```

**See also**

`attributeValue`

## attributeValue

### Syntax

```
XMLnode.attributeValue[ attributeNameOrNumber ]
```

### Description

XML property; returns the value of the specified child node of a parsed XML document.

### Example

Beginning with the following XML:

```
<?xml version="1.0"?>
  <e1>
    <tagName attr1="val1" attr2="val2"/>
    <e2>element 2</e2>
    <e3>element 3</e3>
    here is some text
  </e1>
```

This Lingo returns the value of the first attribute of the tag called `tagName`:

```
put gParserObject.child[1].child[1].attributeValue[1]
-- "val1"
```

### See also

`attributeName`

## audio (RealMedia)

### Syntax

```
sprite(whichSprite).audio
member(whichCastmember).audio
```

### Description

RealMedia sprite or cast member property; allows you to play (TRUE) or mute (FALSE) the audio in the RealMedia stream. The default setting for this property is TRUE (1). Integer values other than 1 or 0 are treated as TRUE (1). Setting this property has no effect if the `realPlayerNativeAudio()` function is set to TRUE.

If the `audio` property is set to FALSE when a RealMedia cast member starts playing, a sound channel is still allocated, which allows you to toggle the sound on and off during playback.

There may be some latency involved in setting this property, which means there may be a slight delay before the sound toggles on or off.

### Examples

The following examples show that the `audio` properties for sprite 2 and the cast member Real is set to TRUE, which means that the audio portion of the RealMedia stream will be played.

```
put sprite(2).audio
-- 1

put member("Real").audio
-- 1
```

The following Lingo sets the `audio` property for sprite 2 and the cast member `Real` to `FALSE`, which means that the audio portion of the `RealMedia` stream will not be played when the movie is played.

```
sprite(2).audio = FALSE  
member("Real").audio = FALSE
```

#### See also

`soundChannel (RealMedia)`, `video (RealMedia)`, `sound`

## auto

### Syntax

```
member(whichCastmember).model(whichModel).lod.auto
```

### Description

3D `lod` modifier property; allows the modifier to manage the reduction of detail in the model as the distance between the model and the camera changes.

The setting of the modifier's `bias` property determines how aggressively the modifier removes detail from the model when the `auto` property is set to `TRUE`.

The modifier updates its `level` property as it adjusts the model's level of detail. Setting the `level` property has no effect unless the `auto` property is set to `FALSE`.

The `#lod` modifier can only be added to models created outside of Director in 3D modeling programs. The value of the `type` property of the model resources used by these models is `#fromFile`. The modifier cannot be added to primitives created within Director.

### Example

This statement sets the `auto` property of the `lod` modifier of the model named `Spaceship` to `TRUE`. The modifier will automatically set the model's level of detail.

```
member("3D World").model("Spaceship").lod.auto = TRUE
```

### See also

`lod (modifier)`, `bias`, `level`

## autoblend

### Syntax

```
member(whichCastmember).model(whichModel).\  
    keyframePlayer.autoblend  
member(whichCastmember).model(whichModel).bonesPlayer.autoblend
```

### Description

3D `keyframePlayer` and `bonesPlayer` modifier property; indicates whether the modifier creates a linear transition to the currently playing motion from the motion that preceded it (`TRUE`) or not (`FALSE`). If `autoBlend` is `TRUE`, the length of the transition is set by the `blendTime` property of the modifier. If `autoBlend` is `FALSE`, the transition is controlled by the `blendFactor` property of the modifier and `blendTime` is ignored.

Motion blending is completely disabled when `blendTime` is set to 0 and `autoBlend` is set to `TRUE`.

The default value of this property is `TRUE`.

### Example

This statement turns `autoblend` off for the model named `Alien3`. The model's `blendFactor` setting will be used for blending successive motions in the playlist.

```
member("newaliens").model("Alien3").keyframePlayer.\
    autoblend = FALSE
```

### See also

`blendFactor`, `blendTime`

## autoCameraPosition

### Syntax

```
member(whichTextCastmember).autoCameraPosition
```

### Description

3D camera property; indicates whether the camera of the 3D text cast member is automatically positioned to show all of the text (TRUE) or not (FALSE). This is useful when changing the text, font, fontsize, and other properties of the cast member.

This property is not valid with other types of 3D cast members.

### Example

This statement sets the `autoCameraPosition` property of the cast member named `Headline` to `FALSE`. When the cast member is displayed in 3D mode, the camera will not be positioned automatically.

```
member("Headline").autoCameraPosition = FALSE
```

### See also

`displayMode`

## autoMask

### Syntax

```
member(whichCursorCastMember).autoMask  
the autoMask of member whichCastMember
```

### Description

Cast member property; specifies whether the white pixels in the animated color cursor cast member *whichCursorCastMember* are transparent, allowing the background to show through (TRUE, default), or opaque (FALSE).

### Example

In this script, when the custom animated cursor stored in cast member 5 enters the sprite, the automask is turned on so that the background of the sprite will show through the white pixels. When the cursor leaves the sprite, the automask is turned off.

Using dot syntax, the script is written as:

```
on mouseEnter  
    member 5.autoMask = TRUE  
end  
  
on mouseLeave  
    member 5.autoMask = FALSE  
end
```

Using traditional Lingo syntax, the script is written as:

```
on mouseEnter
    set the autoMask of member 5 = TRUE
end

on mouseLeave
    set the autoMask of member 5 = FALSE
end
```

## autoTab

### Syntax

```
member(whichCastMember).autoTab
the autoTab of member whichCastMember
```

### Description

Cast member property; determines the effect that pressing the Tab key has on the editable field or text cast member specified by *whichCastMember*. The property can be made active (TRUE) or inactive (FALSE). Tabbing order depends on sprite number order, not position on the Stage.

This property is always TRUE in an applet created with the Save as Java feature of Director.

### Example

This statement causes the cast member Comments to automatically advance the insertion point to the next editable field or text sprite after the user presses Tab.

Dot syntax:

```
member ("Comments").autotab = TRUE
```

Verbose Lingo syntax:

```
set the autoTab of member "Comments" to TRUE
```

## axisAngle

### Syntax

```
member(whichCastmember).model(whichModel).transform.axisAngle
member(whichCastmember).camera(whichCamera).transform.axisAngle
member(whichCastmember).light(whichLight).transform.axisAngle
member(whichCastmember).group(whichGroup).transform.axisAngle
transformReference.axisAngle
```

### Description

3D transform property; describes the transform's rotation as an axis/angle pair.

The *axisAngle* property is a linear list containing a vector (the axis) and a float (the angle). The vector is the axis around which the transform is rotated. The float is the amount, in degrees, of rotation.

The default value of this property is [vector( 1.0000, 0.0000, 0.0000 ), 0.0000].

### Examples

This statement shows the rotation of the model named Mailbox as an *axisAngle*. The model is rotated 145.5 degrees counterclockwise about the y axis.

```
put member("Yard").model("Mailbox").transform.axisAngle
-- [vector( 0.0000, 1.0000, 0.0000 ), -145.5000]
```

**See also**

rotation (transform)

## back

**Syntax**

```
member(whichCastmember).modelResource(whichModelResource).back
```

**Description**

3D #box model resource property; indicates whether the side of the box intersected by its +Z axis is scaled (TRUE) or open (FALSE).

The default value for this property is TRUE.

**Example**

This statement sets the back property of the model resource named Crate to FALSE, meaning the back of this box will be open.

```
member("3D World").modelResource("Crate").back = FALSE
```

**See also**

bottom (3D), front, top (3D), left (3D), right (3D)

## backColor

**Syntax**

```
member(whichCastMember).backColor = colorNumber  
set the backColor of member whichCastMember to colorNumber  
sprite(whichSprite).backColor  
the backColor of sprite whichSprite
```

**Description**

Cast member and sprite property; sets the background color of the specified cast member or sprite according to the color value assigned.

- For cast members: it affects field or button cast member displays.
- For sprites: setting the backColor of a sprite is the same as choosing the background color from the tool palette when the sprite is selected on the Stage. For the value that Lingo sets to last beyond the current sprite, the sprite must be a puppet. The background color applies only to 1-bit bitmap and shape cast members.

The backColor value ranges from 0 to 255 for 8-bit color and from 0 to 15 for 4-bit color. The numbers correspond to the index number of the background color in the current palette. (A color's index number appears in the color palette's lower left corner when you click the color.)

You should not apply this property to bitmap cast members deeper than 1-bit, as the results are difficult to predict.

For a movie that plays back as an applet created with the Save as Java feature of Director, specify colors for backColor using the decimal equivalent of the 24-bit hexadecimal values used in an HTML document.

For example, the hexadecimal value for pure red, FF0000, is equivalent to 16711680 in decimal numbers. This statement specifies pure red as a cast member's background color:

```
set the backColor of member = 16711680
```



This property can be tested and set.

**Note:** It is recommended that the newer `bgColor` property be used instead of the `backColor` property.

### Examples

This statement changes the color of the characters in cast member 1 to the color in palette entry 250.

Dot syntax:

```
member(1).backColor = 250
```

Verbose Lingo syntax:

```
set the backColor of member 1 to 250
```

The following statement sets the variable `oldColor` to the background color of sprite 5:

```
oldColor = sprite (5).backColor
```

The following statement randomly changes the background color of a random sprite between sprites 11 and 13 to color number 36:

```
sprite(10 + random(3)).backColor = 36
```

### See also

`bgColor`, `color` (sprite and cast member property)

## backdrop

### Syntax

```
sprite(whichSprite).camera{( index )}.backdrop[ index ].loc  
member(whichCastmember).camera(whichCamera).backdrop[ index ].loc  
sprite(whichSprite).camera{( index )}.backdrop[ index ].source  
member(whichCastmember).camera(whichCamera).backdrop[ index ].source  
sprite(whichSprite).camera{( index )}.backdrop[ index ].scale  
member(whichCastmember).camera(whichCamera).backdrop[ index ].scale  
sprite(whichSprite).camera{( index )}.backdrop[ index ].rotation  
member(whichCastmember).camera(whichCamera).\  
    backdrop[ index ].rotation  
sprite(whichSprite).camera{( index )}.backdrop[ index ].regPoint  
member(whichCastmember).camera(whichCamera).\  
    backdrop[ index ].regPoint  
sprite(whichSprite).camera{( index )}.backdrop[ index ].blend  
member(whichCastmember).camera(whichCamera).backdrop[ index ].blend  
sprite(whichSprite).camera{( index )}.backdrop.count  
member(whichCastmember).camera(whichCamera).backdrop.count
```

### Description

3D camera property; a 2D image that is rendered on the camera's projection plane. All models in the camera's view appear in front of the backdrop.

Backdrops have the following properties:

**Note:** These properties can also be used to get, set, and manipulate overlays. See the individual property entries for detailed information.

**loc (backdrop and overlay)** indicates the 2D location of the backdrop, as measured from the upper left corner of the sprite.

**source** indicates the texture used by the backdrop.

**scale (backdrop and overlay)** is the number by which the height and width of the texture are multiplied to determine the dimensions of the backdrop.

**rotation (backdrop and overlay)** is the amount by which the backdrop is rotated about its `regPoint`.

**regPoint (3D)** indicates the registration point of the backdrop.

**blend (3D)** indicates the opacity of the backdrop.

**count** indicates the number of items in the camera's list of backdrops.

Use the following commands to create and remove backdrops:

**addBackdrop** creates a backdrop from a texture and adds it to the end of the camera's list of backdrops.

**insertBackdrop** creates a backdrop from a texture and adds it to the camera's list of backdrops at a specific index position.

**removeBackdrop** deletes the backdrop.

**See also**

`overlay`

## backgroundColor

### Syntax

`member(whichVectorMember).backgroundColor`  
the `backgroundColor` of member *whichVectorMember*

### Description

Vector shape cast member property; sets the background color of the specified cast member or sprite to the RGB color value assigned.

This property can be both tested and set.

### Example

```
member("Archie").backgroundColor= rgb(255,255,255)
```

**See also**

`bgColor`

## BACKSPACE

### Syntax

`BACKSPACE`

### Description

Constant; represents the Backspace key. This key is labeled Backspace in Windows and Delete on the Macintosh.

### Example

This `on keyDown` handler checks whether the Backspace key was pressed and, if it was, calls the handler `clearEntry`:

```
on keyDown
  if the key = BACKSPACE then clearEntry
  stopEvent
end keyDown
```

## beep

### Syntax

`beep {numberOfTimes}`

### Description

Command; causes the computer's speaker to beep the number of times specified by *numberOfTimes*. If *numberOfTimes* is missing, the beep occurs once.

- In Windows, the beep is the sound assigned in the Sounds Properties dialog box.
- For the Macintosh, the beep is the sound selected from Alert Sounds on the Sound control panel. If the volume on the Sound control panel is set to 0, the menu bar flashes instead.

### Example

This statement causes two beeps if the Answer field is empty:

```
if field "Answer" = EMPTY then beep 2
```

## beepOn

### Syntax

`the beepOn`

### Description

Movie property; determines whether the computer automatically beeps when the user clicks on anything except an active sprite (TRUE), or not (FALSE, default).

Scripts that set `beepOn` should be placed in frame or movie scripts.

This property can be tested and set.

### Examples

This statement sets `beepOn` to TRUE:

```
the beepOn = TRUE
```

This statement sets `beepOn` to the opposite of its current setting:

```
the beepOn = not the beepOn
```

## before

### See

`put...before`

## beginRecording

### Syntax

`beginRecording`

### Description

Keyword; starts a Score generation session. Only one update session in a movie can be active at a time.

Every `beginRecording` keyword must be matched by an `endRecording` keyword that ends the Score generation session.

### Example

When used in the following handler, the `beginRecording` keyword begins a Score generation session that animates the cast member `Ball` by assigning the cast member to sprite channel 20 and then moving the sprite horizontally and vertically over a series of frames. The number of frames is determined by the argument `numberOfFrames`.

```
on animBall numberOfFrames
  beginRecording
    horizontal = 0
    vertical = 100
    repeat with i = 1 to numberOfFrames
      go to frame i
      sprite(20).member = member "Ball"
      sprite(20).locH = horizontal
      sprite(20).locV = vertical
      sprite(20).type = 1
      sprite(20).foreColor = 255
      horizontal = horizontal + 3
      vertical = vertical + 2
      updateFrame
    end repeat
  endRecording
end
```

### See also

`endRecording`, `updateFrame`, `scriptNum`, `tweened`

## on beginSprite

### Syntax

```
on beginSprite
  statement(s)
end
```

### Description

System message and event handler; contains statements that run when the playhead moves to a frame that contains a sprite that was not previously encountered. Like `endSprite`, this event is generated only one time, even if the playhead loops on a frame, since the trigger is a sprite not previously encountered by the playhead. The event is generated before `prepareFrame`.

Director creates instances of any behavior scripts attached to the sprite when the `beginSprite` message is sent.

The object reference `me` is passed to this event if it is used in a behavior. The message is sent to behaviors and frame scripts.

If a sprite begins in the first frame that plays in the movie, the `beginSprite` message is sent after the `prepareMovie` message but before the `prepareFrame` and `startMovie` messages.

**Note:** Be aware that some sprite properties, such as the `rect` sprite property, may not be accessible in a `beginSprite` handler. This is because the property needs to be calculated, which is not done until the sprite is drawn.

The `go`, `play`, and `updateStage` commands are disabled in an `on beginSprite` handler.

### Example

This handler plays the sound cast member Stevie Wonder when the sprite begins:

```
on beginSprite me
    puppetSound "Stevie Wonder"
end
```

### See also

on endSprite, on prepareFrame, scriptInstanceList

## bevelDepth

### Syntax

```
member(whichTextCastmember).bevelDepth
member(which3DCastmember).modelResource(whichModelResource).\
    bevelDepth
```

### Description

3D text property; indicates the degree of beveling on the 3D text.

For text cast members, this property has no effect unless the member's `displayMode` property is set to `#mode3D` and its `bevelType` property is set to `#miter` or `#round`.

For extruded text in a 3D cast member, this property has no effect unless the model resource's `bevelType` property is set to `#miter` or `#round`.

The range of this property is 0.0 to 10.0, and the default setting is 10.0.

### Example

In this example, the cast member named Logo is a text cast member. This statement sets the `bevelDepth` of logo to 5.5. When logo is displayed in 3D mode, if its `bevelType` property is set to `#miter` or `#round`, the edges of its letters will exhibit dramatic beveling.

```
member("Logo").bevelDepth = 5.5
```

In this example, the model resource of the model named Slogan is extruded text. This statement sets the `bevelDepth` of Slogan's model resource to 5. If the `bevelType` property of Slogan is set to `#miter` or `#round`, the edges of its letters will exhibit dramatic beveling.

```
member("scene").model("Slogan").resource.bevelDepth = 5
```

### See also

`bevelType`, `extrude3D`, `displayMode`

## bevelType

### Syntax

```
member(whichTextCastmember).bevelType
member(which3DCastmember).modelResource(whichModelResource).\
    bevelType
```

### Description

3D text property; indicates the style of beveling applied to the 3D text.

For text cast members, this is a member property. For extruded text in a 3D cast member, this is a model resource property.

The `bevelType` property has the following possible values:

- `#none`
- `#miter` (the default)
- `#round`

#### Example

In this example, the cast member named `Logo` is a text cast member. This statement sets the `bevelType` of `Logo` to `#round`.

```
member("logo").bevelType = #round
```

In this example, the model resource of the model named `Slogan` is extruded text. This statement sets the `bevelType` of `Slogan's` model resource to `#miter`.

```
member("scene").model("Slogan").resource.bevelType = #miter
```

#### See also

`bevelDepth`, `extrude3D`, `displayMode`

## bgColor

#### Syntax

```
sprite(whichSpriteNumber).bgColor  
the bgColor of sprite whichSpriteNumber  
the bgColor of the stage  
(the stage).bgColor  
member(which3dMember).bgcolor
```

#### Description

Sprite property, system property, and 3D cast member property; determines the background color of the sprite specified by *whichSprite*, the color of the Stage, or the background color of the 3D cast member. Setting the `bgColor` sprite property is equivalent to choosing the background color from the Tools window when the sprite is selected on the Stage. Setting the `bgColor` property for the Stage is equivalent to setting the color in the Movie Properties dialog box.

The `sprite` property has the equivalent functionality of the `backColor` sprite property, but the color value returned is a color object of whatever type has been set for that sprite.

This property can be tested and set.

#### Example

This example sets the color of the Stage to an RGB value.

Dot syntax:

```
(the stage).bgColor = rgb(255, 153, 0)
```

Verbose Lingo syntax:

```
set the bgColor of the stage = rgb(255, 153, 0)
```

#### See also

`color()`, `backColor`, `backgroundColor`, `stageColor`

## bias

### Syntax

```
member(whichCastmember).model(whichModel).lod.bias
```

### Description

3D lod modifier property; indicates how aggressively the modifier removes detail from the model when its `auto` property is set to `TRUE`. This property has no effect when the modifier's `auto` property is set to `FALSE`.

The range for this property is from 0.0 (removes all polygons) to +100.0 (removes no polygons). The default setting is 100.0.

The `#lod` modifier can only be added to models created outside of Director in 3D modeling programs. The value of the `type` property of the model resources used by these models is `#fromFile`. The modifier cannot be added to primitives created within Director.

### Example

This statement sets the `bias` property of the `lod` modifier of the model named `Spaceship` to 10. If the `lod` modifier's `auto` property is set to `TRUE`, the modifier will very aggressively lower the level of detail of `Spaceship` as it moves away from the camera.

```
member("3D World").model("Spaceship").lod.bias = 10
```

### See also

`lod` (modifier), `auto`, `level`

## bitAnd()

### Syntax

```
bitAnd(integer1, integer2)
```

### Description

Function; converts the two specified integers to 32-bit binary numbers and returns a binary number whose digits are 1's in the positions where both numbers had a 1, and 0's in every other position. The result is the new binary number, which Lingo displays as a base 10 integer.

Integer	Binary number (abbreviated)
6	00110
7	00111
Result	
6	00110

### Example

This statement compares the binary versions of the integers 6 and 7 and returns the result as an integer:

```
put bitAnd(6, 7)
-- 6
```

### See also

`bitNot()`, `bitOr()`, `bitXor()`

## bitmapSizes

### Syntax

`member(whichFontMember).bitmapSizes`  
the `bitmapSizes` of member *whichFontMember*

### Description

Font cast member property; returns a list of the bitmap point sizes that were included when the font cast member was created.

### Example

This statement displays the bitmap point sizes that were included when cast member 11 was created:

```
put member(11).bitmapSizes
-- [12, 14, 18]
```

### See also

`recordFont`, `characterSet`, `originalFont`

## bitNot()

### Syntax

`(integer).bitNot`  
`bitNot(integer)`

### Description

Function; converts the specified integer to a 32-bit binary number and reverses the value of each binary digit, replacing 1's with 0's and 0's with 1's. The result is the new binary number, which Lingo displays as a base 10 integer.

Integer	Binary number
1	00000000000000000000000000000001
Result	
-2	11111111111111111111111111111110

### Example

This statement inverts the binary representation of the integer 1 and returns a new number.

```
put (1).bitNot
-- -2
```

### See also

`bitAnd()`, `bitOr()`, `bitXor()`



## bitOr()

### Syntax

`bitOr(integer1, integer2)`

### Description

Function; converts the two specified integers to 32-bit binary numbers and returns a binary number whose digits are 1's in the positions where either number had a 1, and 0's in every other position. The result is the new binary number, which Lingo displays as a base 10 integer.

Integer	Binary number (abbreviated)
5	0101
6	0110
<b>Result</b>	
7	0111

### Example

This statement compares the 32-bit binary versions of 5 and 6 and returns the result as an integer:

```
put bitOr(5, 6)
-- 7
```

### See also

`bitNot()`, `bitAnd()`, `bitXor()`

## bitRate

### Syntax

`member(whichCastMember).bitRate`  
the bitRate of member *whichCastMember*

### Description

Shockwave Audio (SWA) cast member property; returns the bit rate, in kilobits per second (Kbps), of the specified SWA cast member that has been preloaded from the server.

The `bitRate` member property returns 0 until streaming begins.

### Example

This behavior outputs the bit rate of an SWA cast member when the sprite is first encountered.

Dot syntax:

```
property spriteNum

on beginSprite me
    memName = sprite(spriteNum).member.name
    put "The bitRate of member"&&memName&&"is"&&member(memName).bitRate
end
```

Verbose syntax:

```
property spriteNum
```

```
on beginSprite me
  memName = sprite(spriteNum).member.name
  put "The bitRate of member"&&memName&&"is"&&member(memName).bitRate
end
```

## bitsPerSample

### Syntax

```
member(whichCastMember).bitsPerSample
the bitsPerSample of member whichCastMember
```

### Description

Shockwave Audio (SWA) cast member property; indicates the bit depth of the original file that has been encoded for Shockwave Audio (SWA). This property is available only after the SWA sound begins playing or after the file has been preloaded using the `preloadBuffer` command.

This property can be tested but not set.

### Example

This statement assigns the original bit rate of the file used in SWA streaming cast member Paul Robeson to the field cast member How Deep.

Dot syntax:

```
put member "Paul Robeson".bitsPerSample into member "How Deep"
```

Verbose syntax:

```
put the bitsPerSample of member "Paul Robeson" into member "How Deep"
```

## bitXor()

### Syntax

```
bitXor(integer1, integer2)
```

### Description

Function; converts the two specified integers to 32-bit binary numbers and returns a binary number whose digits are 1's in the positions where the given numbers' digits do not match, and 0's in the positions where the digits are the same. The result is the new binary number, which Lingo displays as a base 10 integer.

Integer	Binary number (abbreviated)
5	0101
6	0110
<b>Result</b>	
3	0011

### Example

This statement compares the 32-bit binary versions of 5 and 6 and returns the result as an integer:

```
put bitXor(5, 6)
-- 3
```

### See also

`bitNot()`, `bitOr()`, `bitAnd()`

## blend

### Syntax

```
sprite(whichSprite).blend  
the blend of sprite whichSprite
```

### Description

Sprite property; sets or determines a sprite's blend value, from 0 to 100, corresponding to the blend values in the Sprite Properties dialog box.

The possible colors depend on the colors available in the palette, regardless of the monitor's color depth.

The Director player for Java supports the `blend` sprite property for bitmap sprites only.

For best results, use the blend ink with images that have a color depth greater than 8-bit.

### Examples

The following statement sets the blend value of sprite 3 to 40 percent.

Dot syntax:

```
sprite(3).blend = 40
```

Verbose syntax:

```
set the blend of sprite 3 to 40
```

This statement displays the blend value of sprite 3 in the Message window:

```
put the blend of sprite 3
```

### See also

`blendLevel`

## blend (3D)

### Syntax

```
sprite(whichSprite).camera{( index )}.backdrop[ index ].blend  
member(whichCastmember).camera(whichCamera).backdrop[ index ].blend  
sprite(whichSprite).camera{( index )}.overlay[ index ].blend  
member(whichCastmember).camera(whichCamera).overlay[ index ].blend  
member(whichCastmember).shader(whichShader).blend  
member(whichCastmember).model(whichModel).shader.blend  
member(whichCastmember).model(whichModel).shaderList{[ index ]}.blend
```

### Description

3D backdrop, overlay, and `#standard` shader property; indicates the opacity of the backdrop, overlay, or shader.

Setting the `blend` property of a shader will have no effect unless the shader's `transparent` property is set to `TRUE`.

The range of this property is 0 to 100, and the default value is 100.

### Example

This statement sets the `blend` property of the shader for the model named `Window` to 80. If the `transparent` property of `Window`'s shader is set to `TRUE`, the model will be slightly transparent.

```
member("House").model("Window").shader.blend = 80
```

### See also

`bevelDepth`, `overlay`, `shadowPercentage`, `transparent`

## blendConstant

### Syntax

```
member(whichCastmember).shader(whichShader).blendConstant  
member(whichCastmember).model(whichModel).shader.blendConstant  
member(whichCastmember).model(whichModel).shaderList[[index]].\  
    blendConstant
```

### Description

3D `#standard` shader property; indicates the blending ratio used for the first texture layer of the shader.

If the shader's `useDiffuseWithTexture` property is set to `TRUE`, the texture blends with the color set by the shader's `diffuse` property. If `useDiffuseWithTexture` is `FALSE`, white is used for blending.

Each of the other texture layers blends with the texture layer below it. Use the `blendConstantList` property to control blending in those texture layers.

The `blendConstant` property works only when the shader's `blendSource` property is set to `#constant`. For more information, see `blendSource` and `blendSourceList`.

The range of this property is 0 to 100; the default is 50.

### Example

In this example, the shader list of the model named `MysteryBox` contains six shaders. This statement sets the `blendConstant` property of the second shader to 20. This property is affected by the settings of the `blendFunction`, `blendFunctionList`, `blendSource`, and `blendSourceList` properties.

```
member("Level2").model("MysteryBox").shaderList[2].\  
    blendConstant = 20
```

### See also

`blendConstantList`, `blendFunction`, `blendFunctionList`, `blendSource`, `blendSourceList`, `useDiffuseWithTexture`, `diffuse`, `diffuseColor`

# blendConstantList

## Syntax

```
member(whichCastmember).shader(whichShader).blendConstantList
member(whichCastmember).model(whichModel).shader.blendConstant\
List{[index]}
member(whichCastmember).model(whichModel).shaderList{[index]}. \
blendConstantList{[index]}
```

## Description

3D #standard shader property; indicates the ratio used for blending a texture layer of the shader with the texture layer below it.

The shader's texture list and the blend constant list both have eight index positions. Each index position in the blend constant list controls blending for the texture at the corresponding index position in the texture list. You can set all index positions of the list to the same value at one time by not specifying the optional *index* parameter. Use the *index* parameter to set the list one index position at a time.

The `blendConstantList` property works only when the `blendSource` property of the corresponding texture layer is set to #constant.

The range of this property is 0 to 100; the default is 50.

## Example

In this example, the shader list of the model named `MysteryBox` contains six shaders. This statement shows the `blendConstant` property of each of the textures used by the second shader. This property is affected by the settings of the `blendFunction`, `blendFunctionList`, `blendSource`, and `blendSourceList` properties.

```
put member("Level2").model("MysteryBox").shaderList[2]. \
blendConstantList
-- [20.0000, 50.0000, 50.0000, 50.0000, 20.0000, 50.0000, \
50.0000, 50.0000]
```

## See also

`blendConstant`, `blendFunction`, `blendFunctionList`, `blendSource`, `blendSourceList`, `useDiffuseWithTexture`, `diffuse`, `diffuseColor`

# blendFactor

## Syntax

```
member(whichCastmember).model(whichModel).keyframePlayer. \
blendFactor
member(whichCastmember).model(whichModel).bonesPlayer.blendFactor
```

## Description

3D `keyframePlayer` and `bonesPlayer` modifier property; indicates the amount by which a motion is combined with the motion that preceded it.

The range of this property is 0 to 100, and the default value is 0.

`BlendFactor` is used only when the `autoblend` property of the modifier is set to `FALSE`. If the value of the `blendFactor` property is 100, the current motion will have none of the characteristics of the motion that preceded it. If the value of `blendFactor` is 0, the current motion will have all of the characteristics of the motion that preceded it and none of its own. If the value of `blendFactor` is 50, the current motion will be a synthesis equally composed of its own characteristics and those of the motion that preceded it. The value `blendFactor` can be varied over time to create transitions unlike the linear transition created when the modifier's `autoblend` property is set to `TRUE`.

### Example

This statement sets the `blendFactor` property of model `Alien3` to 50. If the modifier's `autoblend` property is `FALSE`, each motion in the playlist of the `keyframePlayer` for `Alien3` will be an even mixture of itself and the motion that preceded it.

```
member("newaliens").model("Alien3").keyframePlayer.blendFactor = 50
```

### See also

`autoblend`, `keyframePlayer (modifier)`

## blendFunction

### Syntax

```
member(whichCastmember).shader(whichShader).blendFunction  
member(whichCastmember).model(whichModel).shader.blendFunction  
member(whichCastmember).model(whichModel).shaderList[[index]].\  
    blendFunction
```

### Description

3D `#standard` shader property; indicates the type of blending used by the first texture layer of the shader.

If the shader's `useDiffuseWithTexture` property is set to `TRUE`, the texture blends with the color set by the shader's `diffuse` property. If `useDiffuseWithTexture` is `FALSE`, white is used for blending.

Each of the other texture layers blends with the texture layer below it. Use the `blendFunctionList` property to control blending in those texture layers.

The `blendFunction` property can have the following values:

`#multiply` multiplies the RGB values of the texture layer by the color being used for blending (see above).

`#add` adds the RGB values of the texture layer to the color being used for blending, and then clamps to 255.

`#replace` prevents the texture from being blended with the color set by the shader's `diffuse` property.

`#blend` combines the colors of the texture layer with the color being used for blending in the ratio set by the `blendConstant` property.

The default value of this property is `#multiply`.

### Example

In this example, the shader list of the model named `MysteryBox` contains six shaders. This statement sets the `blendFunction` property of the second shader to `#blend`. This enables the settings of the `blendSource`, `blendSourceList`, `blendConstant`, and `blendConstantList` properties.

```
member("Level2").model("MysteryBox").shaderList[2].\
    blendFunction = #blend
```

### See also

`blendConstant`, `blendConstantList`, `blendFunctionList`, `blendSource`, `blendSourceList`, `useDiffuseWithTexture`, `diffuse`, `diffuseColor`

## blendFunctionList

### Syntax

```
member(whichCastmember).shader(whichShader).\
    blendFunctionList[[index]]
member(whichCastmember).model(whichModel).shader.\
    blendFunctionList[[index]]
member(whichCastmember).model(whichModel).shaderList[[index]].\
    blendFunctionList[[index]]
```

### Description

3D `#standard` shader property; a linear list that indicates the manner in which each texture layer blends with the texture layer below it.

The shader's texture list and blend function list both have eight index positions. Each index position in the blend function list controls blending for the texture at the corresponding index position in the texture list. You can set all index positions of the list to the same value at one time by not specifying the optional `index` parameter. Use the `index` parameter to set the list one index position at a time.

Each index position of the blend function list can have one of the following values:

`#multiply` multiplies the RGB values of the texture layer by the RGB values of the texture layer below it.

`#add` adds the RGB values of the texture layer to the RGB values of the texture layer below it, and then clamps to 255.

`#replace` causes the texture to cover the texture layer below it. No blending occurs.

`#blend` causes blending to be controlled by the value of the `blendSource` property, which allows alpha blending.

The default value of this property is `#multiply`.

### Example

In this example, the `shaderList` property of the model named `MysteryBox` contains six shaders. This statement shows that the value of the fourth index position of the `blendFunctionList` property of the second shader is set to `#blend`. Blending of the fourth texture layer of the second shader of the model will be controlled by the settings of the `blendSource`, `blendSourceList`, `blendConstant`, `blendConstantList`, `diffuse`, `diffuseColor`, and `useDiffuseWithTexture` properties.

```
put member("Level2").model("MysteryBox").shaderList[2].\
    blendFunctionList[4]
-- #blend
```

### See also

`blendConstant`, `blendConstantList`, `blendFunction`, `blendSource`, `blendSourceList`, `diffuse`, `diffuseColor`, `useDiffuseWithTexture`

## blendLevel

### Syntax

```
sprite(whichSpriteNumber).blendLevel  
the blendLevel of sprite whichSpriteNumber
```

### Description

Sprite property; allows the current blending value of a sprite to be set or accessed. The possible range of values is from 0 to 255. This differs from the Sprite Inspector, which shows values in the range 0 to 100. The results are the same, the scales simply differ.

This property is the equivalent of the `blend` sprite property.

### Example

```
sprite(3).blendlevel = 99
```

### See also

`blend`

## blendRange

### Syntax

```
member(whichCastmember).modelResource(whichModelResource)\  
.blendRange.start  
modelResourceObjectReference.blendRange.end  
member(whichCastmember).modelResource(whichModelResource)\  
.blendRange.start  
modelResourceObjectReference.blendRange.end
```

### Description

3D property; when used with a model resource whose type is `#particle`, allows you to get or set the start and end of the model resource's blend range.

The opacity of particles in the system is interpolated linearly between `blendRange.start` and `blendRange.end` over the lifetime of each particle.

This property's value must be greater than or equal to 0.0 and less than or equal to 100.0. The default value for this property is 100.0.

### Example

This statement sets the `blendRange` properties of model resource `ThermoSystem`, which is of the type `#particle`.

The first line sets the start value to 100, and the second line sets the end value to 0. The effect of this statement is that the particles of `ThermoSystem` are fully opaque when they first appear, and then gradually fade to transparent during their lifetime.

```
member("Heater").modelResource("ThermoSystem").blendRange.\  
start = 100.0  
member("Heater").modelResource("ThermoSystem").blendRange.\  
end = 0.0
```



# blendSource

## Syntax

```
member(whichCastmember).shader(whichShader).blendSource
member(whichCastmember).model(whichModel).shader.blendSource
member(whichCastmember).model(whichModel).shaderList[[index]].\
    blendSource
```

## Description

3D #standard shader property; indicates whether blending of the first texture layer in the shader's texture list is based on the texture's alpha information or a constant ratio.

If the shader's `useDiffuseWithTexture` property is set to `TRUE`, the texture blends with the color set by the shader's `diffuse` property. If `useDiffuseWithTexture` is `FALSE`, white is used for blending.

Each of the other texture layers blends with the texture layer below it. Use the `blendSourceList` property to control blending in those texture layers.

The `blendSource` property works only when the shader's `blendFunction` property is set to `#blend`.

The possible values of this property are as follows:

`#alpha` causes the alpha information in the texture to determine the blend ratio of each pixel of the texture with the color being used for blending (see above).

`#constant` causes the value of the shader's `blendConstant` property to be used as the blend ratio for all the pixels of the texture.

The default value of this property is `#constant`.

## Example

In this example, the shader list of the model named `MysteryBox` contains six shaders. This statement sets the `blendSource` property of the first texture used by the second shader to `#constant`. This enables the settings of the `blendConstant` and `blendConstantList` properties.

```
member("Level2").model("MysteryBox").shaderList[2].\
    blendSource = #constant
```

## See also

`blendSourceList`, `blendFunction`, `blendFunctionList`, `blendConstant`,  
`blendConstantList`, `useDiffuseWithTexture`, `diffuse`, `diffuseColor`

# blendSourceList

## Syntax

```
member(whichCastmember).shader(whichShader).\
    blendSourceList[index]
member(whichCastmember).model(whichModel).shader.\
    blendSourceList[[index]]
member(whichCastmember).model(whichModel).\
    shaderList[[index]].blendSourceList[[index]]
```

### Description

3D `#standard` shader property; indicates whether blending of a texture layer with the texture layers below it is based on the texture's alpha information or a constant ratio.

The shader's texture list and the blend source list both have eight index positions. Each index position in the blend source list controls blending for the texture at the corresponding index position in the texture list. You can set all index positions of the list to the same value at one time by not specifying the optional *index* parameter. Use the *index* parameter to set the list one index position at a time.

The `blendSourceList` property only works when the `blendFunction` property of the corresponding texture layer is set to `#blend`. See `blendFunction` and `blendFunctionList` for more information.

The possible values of this property are as follows:

`#alpha` causes the alpha information in the texture to determine the blend ratio of each pixel of the texture layer with the layer below it.

`#constant` causes the value of the `blendConstant` property of the corresponding texture layer to be used as the blend ratio for all of the pixels of the texture layer. See `blendConstant` and `blendConstantList` for more information.

The default value of this property is `#constant`.

### Example

In this example, the shader list of the model `MysteryBox` contains six shaders. Each shader has a texture list that contains up to eight textures. This statement shows that the `blendSource` property of the fourth texture used by the second shader is set to `#constant`. This enables the settings of the `blendConstant`, `blendConstantList`, and `useDiffuseWithTexture` properties.

```
member("Level2").model("MysteryBox").shaderList[2].\
    blendSourceList[4] = #constant
```

### See also

`blendSource`, `blendFunction`, `blendFunctionList`, `blendConstant`, `blendConstantList`, `useDiffuseWithTexture`, `diffuse`, `diffuseColor`

## blendTime

### Syntax

```
member(whichCastmember).model(whichModel).keyframePlayer.\
    blendTime
member(whichCastmember).model(whichModel).bonesPlayer.blendTime
```

### Description

3D `keyframePlayer` and `bonesPlayer` modifier property; determines the duration, in milliseconds, of the transition between motions in the playlist of the modifier for the model.

The `blendTime` property works in conjunction with the modifier's `autoBlend` property. When `autoBlend` is set to `TRUE`, the modifier creates a linear transition to the model's currently playing motion from the motion that preceded it. The value of the `blendTime` property is the length of that transition. The `blendTime` property is ignored if `autoBlend` is set to `FALSE`.

The default setting of this property is 500.

### Example

This statement sets the length of the transition between motions in the playlist of the modifier for the model named Alien5 to 1200 milliseconds.

```
member("newaliens").model("Alien5").keyframePlayer.\
    blendTime = 1200
```

### See also

autoblend, blendFactor

## bone

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
    bone.count
member(whichCastmember).model(whichModel).bonesPlayer.\
    bone[index].transform
member(whichCastmember).model(whichModel).bonesPlayer.\
    bone[index].worldTransform
```

### Description

3D element; a bone is structural element of a model resource authored in a 3D modeling program. Bones cannot be created, deleted, or rearranged in Director.

Bones (*#bones*) motions, which also must be scripted in a 3D modeling program, act upon the bone structure of a model resource, and are managed in Director by the `bonesPlayer` modifier.

See the entries for `count`, `bonesPlayer` (modifier), `transform` (property), and `worldTransform` for more details.

### See also

`count`, `bonesPlayer` (modifier), `transform` (property), `worldTransform`

## bonesPlayer (modifier)

### Syntax

```
member(whichCastmember).model(whichModel).\
    bonesPlayer.whichBonesPlayerProperty
```

### Description

3D modifier; manages the use of motions by models. The motions managed by the `bonesPlayer` modifier animate segments, called bones, of the model.

Motions and the models that use them must be created in a 3D modeling program, exported as W3D files, and then imported into a movie. Motions cannot be applied to model primitives created within Director.

Adding the `bonesPlayer` modifier to a model by using the `addModifier` command allows access to the following `bonesPlayer` modifier properties:

`playing` (3D) indicates whether a model is executing a motion.

`playlist` is a linear list of property lists containing the playback parameters of the motions that are queued for a model.

`currentTime` (3D) indicates the local time, in milliseconds, of the currently playing or paused motion.

`playRate` is a number that is multiplied by the *scale* parameter of the `play()` or `queue()` command to determine the playback speed of the motion.

`playlist.count` returns the number of motions currently queued in the playlist.

`rootLock` indicates whether the translational component of the motion is used or ignored.

`currentLoopState` indicates whether the motion plays once or repeats continuously.

`blendTime` indicates the length of the transition created by the modifier between motions when the modifier's `autoblend` property is set to `TRUE`.

`autoblend` indicates whether the modifier creates a linear transition to the currently playing motion from the motion that preceded it.

`blendFactor` indicates the degree of blending between motions when the modifier's `autoBlend` property is set to `FALSE`.

`bone[boneId].transform` indicates the transform of the bone relative to the parent bone. You can find the `boneId` value by testing the `getBoneID` property of the model resource. When you set the transform of a bone, it is no longer controlled by the current motion, and cannot be returned to the control of the motion. Manual control ends when the current motion ends.

`bone[boneId].getWorldTransform` returns the world-relative transform of the bone.

`lockTranslation` indicates whether the model can be displaced from the specified planes.

`positionReset` indicates whether the model returns to its starting position after the end of a motion or each iteration of a loop.

`rotationReset` indicates the rotational element of a transition from one motion to the next, or the looping of a single motion.

**Note:** For more detailed information about these properties, see the individual property entries.

The `bonesPlayer` modifier uses the following commands:

`pause()` (3D) halts the motion currently being executed by the model.

`play()` (3D) initiates or unpauses the execution of a motion.

`playNext()` (3D) initiates playback of the next motion in the playlist.

`queue()` (3D) adds a motion to the end of the playlist.

The `bonesPlayer` modifier generates the following events, which are used by handlers declared in the `registerForEvent()` and `registerScript()` commands. The call to the declared handler includes three arguments: the event type (either `#animationStarted` or `#animationEnded`), the name of the motion, and the current time of the motion. For detailed information about notification events, see `registerForEvent()`.

`#animationStarted` is sent when a motion begins playing. If blending is used between motions, the event is sent when the transition begins.

`#animationEnded` is sent when a motion ends. If blending is used between motions, the event is sent when the transition ends.

#### See also

`keyframePlayer (modifier), addModifier, modifiers, modifier`

## border

### Syntax

`member(whichFieldCastmember).border`  
the border of member *whichFieldCastmember*

### Description

Field cast member property; indicates the width, in pixels, of the border around the specified field cast member.

### Example

This statement makes the border around the field cast member Title 10 pixels wide.

Dot syntax:

```
member("Title").border = 10
```

Verbose syntax:

```
set the border of member "Title" to 10
```

## bottom

### Syntax

`sprite(whichSprite).bottom`  
the bottom of sprite *whichSprite*

### Description

Sprite property; specifies the bottom vertical coordinate of the bounding rectangle of the sprite specified by *whichSprite*.

When a movie plays back as an applet, this property's value is measured from the upper left corner of the applet.

This property can be tested and set.

### Example

This statement assigns the vertical coordinate of the bottom of the sprite numbered (i + 1) to the variable named `lowest`.

Dot syntax:

```
set lowest = sprite (i + 1).bottom
```

Verbose syntax:

```
set lowest = the bottom of sprite (i + 1)
```

### See also

`height`, `left`, `locH`, `locV`, `right`, `top`, `width`

## bottom (3D)

### Syntax

`member(whichCastmember).modelResource(whichModelResource).bottom`

**Description**

3D #box model resource property; indicates whether the side of the box intersected by its -Y axis is sealed (TRUE) or open (FALSE).

The default value for this property is TRUE.

**Example**

This statement sets the `bottom` property of the model resource named `GiftBox` to `TRUE`, meaning the bottom of this box will be closed.

```
member("3D World").modelResource("GiftBox").bottom = TRUE
```

**See also**

`back`, `front`, `top` (3D), `left` (3D), `right` (3D), `bottomCap`

## bottomCap

**Syntax**

```
member(whichCastmember).modelResource(whichModelResource).\
    bottomCap
```

**Description**

3D #cylinder model resource property; indicates whether the end of the cylinder intersected by its -Y axis is sealed (TRUE) or open (FALSE).

The default value for this property is TRUE.

**Example**

This statement sets the `bottomCap` property of the model resource named `Tube` to `FALSE`, meaning the bottom of this cylinder will be open.

```
member("3D World").modelResource("Tube").bottomCap = FALSE
```

**See also**

`topCap`, `bottomRadius`, `bottom` (3D)

## bottomRadius

**Syntax**

```
member(whichCastmember).modelResource(whichModelResource).\
    bottomRadius
```

**Description**

3D #cylinder model resource property; indicates the radius, in world units, of the end of the cylinder that is intersected by its -Y axis.

The default value for this property is 25.

**Example**

This statement sets the `bottomRadius` property of the model resource named `Tube` to 38.5.

```
member("3D World").modelResource("Tube").bottomRadius = 38.5
```

**See also**

`topRadius`, `bottomCap`

## bottomSpacing

### Syntax

*chunkExpression*.bottomSpacing

### Description

Text cast member property; enables you to specify additional spacing applied to the bottom of each paragraph in the *chunkExpression* portion of the text cast member.

The value itself is an integer, where less than 0 indicates less spacing between paragraphs and greater than 0 indicates more spacing between paragraphs.

The default value is 0, which results in default spacing between paragraphs.

**Note:** This property, like all text cast member properties, supports only dot syntax.

### Example

This example adds spacing after the first paragraph in cast member News Items.

```
member("News Items").paragraph[1].bottomSpacing=20
```

### See also

top (3D)

## boundary

### Syntax

```
member(whichCastmember).model(whichModel).inker.boundary  
member(whichCastmember).model(whichModel).toon.boundary
```

### Description

3D inker and toon modifier property; allows you to set whether a line is drawn at the edges of a model.

The default setting for this property is TRUE.

### Example

This statement sets the boundary property of the inker modifier applied to the model named Box to TRUE. Lines will be drawn at the edges of the surface of the model.

```
member("shapes").model("Box").inker.boundary = TRUE
```

### See also

lineColor, lineOffset, silhouettes, creases

## boundingSphere

### Syntax

```
member(whichCastmember).model(whichModel).boundingSphere  
member(whichCastmember).group(whichGroup).boundingSphere  
member(whichCastmember).light(whichLight).boundingSphere  
member(whichCastmember).camera(whichCamera).boundingSphere
```

### Description

3D model, group, light, and camera property; describes a sphere that contains the model, group, light, or camera and its children.

The value of this property is a list containing the vector position of the center of the sphere and the floating-point length of the sphere's radius.

This property can be tested but not set.

#### Example

This example displays the bounding sphere of a light in the message window.

```
put member("newAlien").light[5].boundingSphere
-- [vector(166.8667, -549.6362, 699.5773), 1111.0039]
```

#### See also

debug

## boxDropShadow

#### Syntax

member(*whichCastMember*).boxDropShadow  
the boxDropShadow of member *whichCastMember*

#### Description

Cast member property; determines the size, in pixels, of the drop shadow for the box of the field cast member specified by *whichCastMember*.

#### Example

This statement makes the drop shadow of field cast member Title 10 pixels wide.

Dot syntax:

```
member("Title").boxDropShadow = 10
```

Verbose syntax:

```
set the boxDropShadow of member "Title" to 10
```

## boxType

#### Syntax

member(*whichCastMember*).boxType  
the boxType of member *whichCastMember*

#### Description

Cast member property; determines the type of text box used for the specified cast member. The possible values are #adjust, #scroll, #fixed, and #limit.

#### Example

This statement makes the box for field cast member Editorial a scrolling field.

Dot syntax:

```
member("Editorial").boxType = #scroll
```

Verbose syntax:

```
set the boxType of member "Editorial" to #scroll
```



## breakLoop()

### Syntax

```
sound(channelNum).breakLoop()
```

### Description

This function causes the currently looping sound in channel `channelNum` to stop looping and play through to its `endTime`. If there is no current loop, this function has no effect.

### Example

This handler causes the background music looping in sound channel 2 to stop looping and play through to its end:

```
on continueBackgroundMusic  
    sound(2).breakLoop()  
end
```

### See also

`end`, `loopCount`, `loopEndTime`, `loopsRemaining`, `loopStartTime`

## brightness

### Syntax

```
member(whichCastmember).shader(whichShader).brightness  
member(whichCastmember).model(whichModel).shader.brightness  
member(whichCastmember).model(whichModel).shaderList[[index]].\  
    brightness
```

### Description

3D `#newsprint` and `#engraver` shader property; indicates the amount of white blended into the shader.

The range of this property is 1 to 100; the default value is 0.

### Example

This statement sets the brightness of the shader used by the model named `gbCyl2` to half of its maximum value.

```
member("scene").model("gbCyl2").shader.brightness = 50
```

### See also

`newShader`

## broadcastProps

### Syntax

```
member(whichVectorOrFlashMember).broadcastProps  
the broadcastProps of member whichVectorOrFlashMember
```

### Description

Cast member property; controls whether changes made to a Flash or Vector shape cast member are immediately broadcast to all of its sprites currently on the Stage (TRUE) or not (FALSE).

When this property is set to FALSE, changes made to the cast member are used only as defaults for new sprites and don't affect sprites on the Stage.

The default value for this property is TRUE, and it can be both tested and set.

### Example

This frame script assumes that a Flash movie cast member named Navigation Movie has been set up with its `broadcastProps` property set to `FALSE`. The script momentarily allows changes to a Flash movie cast member to be broadcast to its sprites currently on the Stage. It then sets the `viewScale` property of the Flash movie cast member, and that change is broadcast to its sprite. The script then prevents the Flash movie from broadcasting changes to its sprites.

Dot syntax:

```
on enterFrame
    member("Navigation Movie").broadcastProps = TRUE
    member("Navigation Movie").viewScale = 200
    member("Navigation Movie").broadcastProps = FALSE
end
```

Verbose syntax:

```
on enterFrame
    set the broadcastProps of member "Navigation Movie" = TRUE
    set the viewScale of member "Navigation Movie" = 200
    set the broadcastProps of member "Navigation Movie" = FALSE
end
```

## browserName()

### Syntax

```
browserName pathName
browserName()
browserName(#enabled, trueOrFalse)
```

### Description

System property, command, and function; specifies the path or location of the browser. You can use the FileIO Xtra to display a dialog box that allows the user to search for a browser. The `displayOpen()` method of the FileIO Xtra is useful for displaying an Open dialog box.

The form `browserName()` returns the name of the currently specified browser. Placing a `pathname`, like one found using the FileIO Xtra, as an argument in the form `browserName(fullPathToApplication)` allows the property to be set. The form `browserName(#enabled, trueOrFalse)` determines whether the specified browser launches automatically when the `goToNetPage` command is issued.

This command is only useful playing back in a projector or in Director, and has no effect when playing back in a browser.

This property can be tested and set.

### Examples

This statement refers to the location of the Netscape browser:

```
browserName "My Disk:My Folder:Netscape"
```

This statement displays the browser name in a Message window:

```
put browserName()
```

## bufferSize

### Syntax

`member(whichFlashMember).bufferSize`  
the `bufferSize` of member *whichFlashMember*

### Description

Flash cast member property; controls how many bytes of a linked Flash movie are streamed into memory at one time. The `bufferSize` member property can have only integer values. This property has an effect only when the cast member's `preload` property is set to `FALSE`.

This property can be tested and set. The default value is 32,768 bytes.

### Example

This `startMovie` handler sets up a Flash movie cast member for streaming and then sets its `bufferSize` property.

#### Dot syntax:

```
on startMovie
  member("Flash Demo").preload = FALSE
  member("Flash Demo").bufferSize = 65536
end
```

#### Verbose syntax:

```
on startMovie
  set the preload of member "Flash Demo" = FALSE
  set the bufferSize of member "Flash Demo" = 65536
end
```

### See also

`bytesStreamed`, `preLoadRAM`, `stream`, `streamMode`

## build()

### Syntax

`member(whichCastmember).modelResource(whichModelResource).build()`

### Description

3D mesh command; constructs a mesh. This command is only used with model resources whose type is `#mesh`.

You must use the `build()` command in the initial construction of the mesh, after changing any of the face properties of the mesh, and after using the `generateNormals()` command.

### Example

This example creates a simple model resource whose type is `#mesh`, specifies its properties, and then creates a new model using the model resource. The process is outlined in the following line-by-line explanation of the example code:

Line 1 creates a mesh called `Plane`, which has one face, three vertices, and a maximum of three colors. The number of normals and the number of texture coordinates are not set. The normals are created by the `generateNormals` command.

Line 2 defines the vectors that will be used as the vertices for `Plane`.

Line 3 assigns the vectors to the vertices of the first face of `Plane`.

Line 4 defines the three colors allowed by the `newMesh` command.

Line 5 assigns the colors to the first face of Plane. The third color in the color list is applied to the first vertex of Plane, the second color to the second vertex, and the first color to the third vertex. The colors will spread across the first face of Plane in gradients.

Line 6 creates the normals of Plane with the `generateNormals()` command.

Line 7 calls the `build()` command to construct the mesh.

```
nm = member("Shapes").newMesh("Plane",1,3,0,3,0)
nm.vertexList = [vector(0,0,0), vector(20,0,0), vector(20, 20, 0)]
nm.face[1].vertices = [1,2,3]
nm.colorList = [rgb(255,255,0), rgb(0, 255, 0), rgb(0,0,255)]
nm.face[1].colors = [3,2,1]
nm.generateNormals(#smooth)
nm.build()
nm = member("Shapes").newModel("TriModel", nm)
```

#### See also

`generateNormals()`, `newMesh`, `face`

## buttonsEnabled

### Syntax

`sprite(whichFlashSprite).buttonsEnabled`  
the `buttonsEnabled` of sprite *whichFlashSprite*  
`member(whichFlashMember).buttonsEnabled`  
the `buttonsEnabled` of member *whichFlashMember*

### Description

Flash cast member property and sprite property; controls whether the buttons in a Flash movie are active (TRUE, default) or inactive (FALSE). Button actions are triggered only when the `actionsEnabled` property is set to TRUE.

This property can be tested and set.

### Example

This handler accepts a sprite reference and toggles the sprite's `buttonsEnabled` property on or off.

Dot syntax:

```
on ToggleButtons whichSprite
    sprite(whichSprite).buttonsEnabled = not sprite(whichSprite).buttonsEnabled
end
```

Verbose syntax:

```
on ToggleButtons whichSprite
    set the buttonsEnabled of sprite whichSprite = not the buttonsEnabled of \
    sprite whichSprite
end
```

### See also

`actionsEnabled`

# buttonStyle

## Syntax

the `buttonStyle`

## Description

Movie property; determines the visual response of buttons when the user rolls the pointer off them. This property applies only to buttons created with the Button tool in the Tool palette.

The `buttonStyle` property can have these values:

- 0 (list style: default)—Subsequent buttons are highlighted when the pointer passes over them. Releasing the mouse button activates the script associated with that button.
- 1 (dialog style)—Only the first button clicked is highlighted. Subsequent buttons are not highlighted. Releasing the mouse button while the pointer is over a button other than the original button clicked does not activate the script associated with that button.

This property can be tested and set in any type of script.

## Examples

The following statement sets the `buttonStyle` property to 1:

```
the buttonStyle = 1
```

This statement remembers the current setting of the `buttonStyle` property by putting the current `buttonStyle` value in the variable `buttonStyleValue`:

```
buttonStyleValue = the buttonStyle
```

## See also

`checkBoxAccess`, `checkBoxType`

# buttonType

## Syntax

`member(whichCastMember).buttonType`  
the `buttonType` of member *whichCastMember*

## Description

Button cast member property; indicates the specified button cast member's type. Possible values are `#pushButton`, `#checkBox`, and `#radioButton`. This property applies only to buttons created with the button tool in the Tool palette.

## Example

This statement makes the button cast member Editorial a check box.

Dot syntax:

```
member("Editorial").buttonType = #checkBox
```

Verbose syntax:

```
set the buttonType of member "Editorial" to #checkBox
```

## bytesStreamed

### Syntax

member(*whichFlashOrSWAMember*).bytesStreamed  
the bytesStreamed of member *whichFlashOrSWAMember*

### Description

Flash and Shockwave Audio cast member property; indicates the number of bytes of the specified cast member that have been loaded into memory. The `bytesStreamed` property returns a value only when the Director movie is playing. It returns an integer value.

This property can be tested but not set.

### Example

This handler accepts a cast member reference as a parameter, and it then uses the `stream` command to load the cast member into memory. Every time it streams part of the cast member into memory, it uses the `bytesStreamed` property to report in the Message window how many bytes have been streamed.

Dot syntax:

```
on fetchMovie whichFlashMovie
  repeat while member(whichFlashMovie).percentStreamed < 100
    stream(member whichFlashMovie)
    put "Number of bytes streamed:" && member(whichFlashMovie).bytesStreamed
  end repeat
end
```

Verbose syntax:

```
on fetchMovie whichFlashMovie
  repeat while the percentStreamed of member whichFlashMovie < 100
    stream(member whichFlashMovie)
    put "Number of bytes streamed:" && the bytesStreamed of member \
      whichFlashMovie
  end repeat
end
```

### See also

`bufferSize`, `percentStreamed`, `stream`

## bytesStreamed (3D)

### Syntax

member(*whichCastMember*).bytesStreamed

### Description

3D cast member property; indicates how much of the initial file import or the last requested file load has loaded.

### Example

This statement shows that 325,300 bytes of the cast member named Scene have been loaded.

```
put member("Scene").bytesStreamed
-- 325300
```

### See also

`streamSize (3D)`, `state (3D)`

## cacheDocVerify()

### Syntax

```
cacheDocVerify #setting  
cacheDocVerify()
```

### Description

Function; sets how often the contents of a page on the Internet are refreshed with information from the projector's cache. Possible values are *#once* (default) and *#always*.

Specifying *#once* tells a movie to get a file from the Internet once and then use the file from the cache without looking for an updated version on the Internet.

Specifying *#always* tells a movie to try to get an updated version of the file each time the movie calls a URL.

The form `cacheDocVerify()` returns the current setting of the cache.

The `cacheDocVerify` function is valid only for movies running in Director or as projectors. This function is not valid for Shockwave movies because they use the network settings of the browser in which they run.

```
on resetCache  
    current = cacheDocVerify()  
    if current = #once then  
        alert "Turning cache verification on"  
        cacheDocVerify #always  
    end if  
end
```

### See also

`cacheSize()`, `clearCache`

## cacheSize()

### Syntax

```
cacheSize Size  
cacheSize()
```

### Description

Function and command; sets the cache size of Director. The value is in kilobytes.

The `cacheSize` function is valid only for movies running in Director or as projectors. This function is not valid for Shockwave movies because they use the network settings of the browser in which they run.

### Example

This handler checks whether the browser's cache setting is less than 1 MB. If it is, the handler displays an alert and sets the cache size to 1 MB:

```
on checkCache if  
    cacheSize()<1000 then  
        alert "increasing cache to 1MB"  
        cacheSize 1000  
    end if  
end
```

### See also

`cacheDocVerify()`, `clearCache`

# call

## Syntax

```
call #handlerName, script, {args...}
call (#handlerName, scriptInstance, {args...})
```

## Description

Command; sends a message that invokes a handler in specified scripts where *handlerName* is the name of the handler to be activated, *script* references the script or a list of scripts, and *args* are any optional parameters to be passed to the handler.

If *script* is a single script instance, an error alert occurs if the handler is not defined in the script's ancestor script.

If *script* is a list of script instances, the message is sent to each item in the list in turn; if the handler is not defined in the ancestor script, no alert is generated.

The `call` command can use a variable as the name of the handler. Messages passed using `call` are not passed to other scripts attached to the sprite, cast member scripts, frame scripts, or movie scripts.

## Examples

This handler sends the message `bumpCounter` to the first behavior script attached to sprite 1:

```
on mouseDown me
  -- get the reference to the first behavior of sprite 1
  set xref = getAt (the scriptInstanceList of sprite 1,1)
  -- run the bumpCounter handler in the referenced script,
  -- with a parameter
  call (#bumpCounter, xref, 2)
end
```

The following example shows how a `call` statement can call handlers in a behavior or parent script and its ancestor.

- This is the parent script:

```
-- script Man
property ancestor
on new me
  set ancestor = new(script "Animal", 2)
  return me
end
on run me, newTool
  put "Man running with "&the legCount of me&" legs"
end
```

- This is the ancestor script:

```
-- script Animal
property legCount
on new me, newLegCount
  set legCount = newLegCount
  return me
end
on run me
  put "Animal running with "& legCount &" legs"
end
on walk me
  put "Animal walking with "& legCount &" legs"
end
```



- The following statements use the parent script and ancestor script.

This statement creates an instance of the parent script:

```
set m = new(script "man")
```

This statement makes the man walk:

```
call #walk, m
-- "Animal walking with 2 legs"
```

This statement makes the man run:

```
set msg = #run
call msg, m
-- "Man running with 2 legs and rock"
```

This statement creates a second instance of the parent script:

```
set m2 = new(script "man")
```

This statement sends a message to both instances of the parent script:

```
call msg, [m, m2]
-- "Man running with 2 legs "
-- "Man running with 2 legs "
```

## callAncestor

### Syntax

```
callAncestor handlerName, script, {args...}
```

### Description

Command; sends a message to a child object's ancestor script, where *handlerName* is the name of the handler to be activated, *script* references the script instance or a list of script instances, and *args* are any optional parameters to be passed to the handler.

If *script* is a single script instance, an error alert occurs if the handler is not defined in the ancestor of the script.

If *script* is a list of script instances, the message is sent to each item in the list in turn. In this case, if the handler is not defined in an ancestor script, no alert is generated.

Ancestors can, in turn, have their own ancestors.

When you use `callAncestor`, the name of the handler can be a variable, and you can explicitly bypass the handlers in the primary script and go directly to the ancestor script.

### Example

This example shows how a `callAncestor` statement can call handlers in the ancestor of a behavior or parent script.

- This is the parent script:

```
-- script "man"
property ancestor
on new me, newTool
    set ancestor = new(script "Animal", 2)
    return me
end
on run me
    put "Man running with "&the legCount of me&"legs"
end
```

- This is the ancestor script:

```
-- script "animal"
property legCount
on new me, newLegCount
    set legCount = newLegCount
    return me
end
on run me
    put "Animal running with "& legCount &" legs"
end
on walk me
    put "Animal walking with "& legCount &" legs"
end
```

- The following statements use the parent script and ancestor script.

This statement creates an instance of the parent script:

```
set m = new(script "man")
```

This statement makes the man walk:

```
call #walk, m
-- "Animal walking with 2 legs"
```

This statement makes the man run:

```
set msg = #run
callAncestor msg, m
-- "Animal running with 2 legs"
```

This statement creates a second instance of the parent script:

```
set m2 = new(script "man")
```

This statement sends a message to the ancestor script for both men:

```
callAncestor #run,[m,m2]
-- "Animal running with 2 legs"
-- "Animal running with 2 legs"
```

#### See also

`ancestor`, `new()`

## callFrame()

### Syntax

```
sprite(whichSprite).callFrame("FlashLabel")
sprite(whichSprite).callFrame(FlashFrameNumber)
```

### Definition

Command; used to call a series of actions that reside in a frame of a Flash movie sprite. You can specify which frame to call using a frame number or a label. This command sends a message to the Flash ActionScript engine and triggers the actions to execute in the Flash movie.

### Example

This Lingo executes the actions that are attached to frame 10 of the Flash movie in sprite 1:

```
sprite(1).callFrame(10)
```

## camera

### Syntax

```
member(whichCastMember).camera(whichCamera)  
member(whichCastMember).camera[index]  
member(whichCastMember).camera(whichCamera).whichCameraProperty  
member(whichCastMember).camera[index].whichCameraProperty  
sprite(whichSprite).camera{( index )}  
sprite(whichSprite).camera{( index )}.whichCameraProperty
```

### Description

3D element; an object at a vector position from which the 3D world is viewed.

Each sprite has a list of cameras. The view from each camera in the list is displayed on top of the view from camera with lower *index* positions. You can set the `rect (camera)` property of each camera to display multiple views within the sprite.

Cameras are stored in the camera palette of the cast member. Use the `newCamera` and `deleteCamera` commands to create and delete cameras in a 3D cast member.

The `camera` property of a sprite is the first camera in the list of cameras of the sprite. The camera referred to by `sprite(whichSprite).camera` is the same as `sprite(whichSprite).camera(1)`. Use the `addCamera` and `deleteCamera` commands to build the list of cameras in a 3D sprite.

For a complete list of camera properties and commands, see Chapter 2, “3D Lingo by Feature,” on page 31.

### Examples

This statement sets the camera of sprite 1 to the camera named `TreeCam` of the cast member named `Picnic`.

```
sprite(1).camera = member("Picnic").camera("TreeCam")
```

This statement sets the camera of sprite 1 to camera 2 of the cast member named `Picnic`.

```
sprite(1).camera = member("Picnic").camera[2]
```

### See also

`bevelDepth`, `overlay`, `modelUnderLoc`, `spriteSpaceToWorldSpace`, `fog`, `clearAtRender`

## cameraCount()

### Syntax

```
sprite(whichSprite).cameraCount()
```

### Description

3D command; returns the number items in the list of cameras of the sprite.

### Example

This statement shows that sprite 5 contains three cameras.

```
put sprite(5).cameraCount()  
-- 3
```

### See also

`addCamera`, `deleteCamera`

## cameraPosition

### Syntax

```
member(whichCastMember).cameraPosition  
sprite(whichSprite).cameraPosition
```

### Description

3D cast member and sprite property; indicates the position of the default camera.

The default value of this property is vector(0, 0, 250). This is the position of the default camera in a newly created 3D cast member.

### Example

This statement shows that the position of the default camera of the cast member named Babyland is the vector (-117.5992, -78.9491, 129.0254).

```
member("Babyland").cameraPosition = vector(-117.5992, \  
-78.9491, 129.0254)
```

### See also

cameraRotation, autoCameraPosition

## cameraRotation

### Syntax

```
member(whichCastMember).cameraRotation  
sprite(whichSprite).cameraRotation
```

### Description

3D cast member and sprite property; indicates the position of the default camera.

The default value of this property is vector(0, 0, 0). This is the rotation of the default camera in a newly created 3D cast member.

### Example

This statement shows that the rotation of the default camera of the cast member named Babyland is the vector (82.6010, -38.8530, -2.4029).

```
member("babyland").cameraRotation = vector(82.6010, \  
-38.8530, -2.4029)
```

### See also

cameraPosition, autoCameraPosition

## cancelIdleLoad

### Syntax

```
cancelIdleLoad loadTag
```

### Description

Command; cancels the loading of all cast members that have the specified load tag.

### Example

This statement cancels the loading of cast members that have an idle load tag of 20:

```
cancelIdleLoad 20
```

## See also

idleLoadTag

# case

## Syntax

```
case expression of
  expression1 : Statement
  expression2 :
    multipleStatements
  .
  .
  .
  expression3, expression4 :
    Statement
{otherwise:
  statement(s)}
end case
```

## Description

**Keyword;** starts a multiple branching logic structure that is easier to write than repeated `if...then statements`.

Lingo compares the value in `case expression` to the expressions in the lines beneath it, starting at the beginning and continuing through each line in order, until Lingo encounters an expression that matches `case expression`.

When Lingo finds a matching expression, it executes the corresponding statement or statements that follow the colon after the matching expression. When only one statement follows the matching expression, the matching expression and its corresponding statement may appear on the same line. Multiple statements must appear on indented lines immediately below the matching expression.

When more than one possible match could cause Lingo to execute the same statements, the expressions must be separated by commas. (The syntax line containing `expression3` and `expression4` is an example of such a situation.)

After Lingo encounters the first match, it stops testing for additional matches.

If the optional `otherwise` statement is included at the end of the case structure, the statements following `otherwise` are executed if there are no matches.

If a `case` statement tests cases that aren't all integer constants, the Export Xtra for Java converts the `case` statement to an `if...then` statement.

## Examples

The following handler tests which key the user pressed most recently and responds accordingly.

- If the user pressed A, the movie goes to the frame labeled Apple.
- If the user pressed B or C, the movie performs the specified transition and then goes to the frame labeled Oranges.

- If the user pressed any other key, the computer beeps.

```
on keyDown
  case (the key) of
    "A": go to frame "Apple"
    "B", "C":
      puppetTransition 99
      go to frame "Oranges"
    otherwise beep
  end case
end keyDown
```

This `case` statement tests whether the cursor is over sprite 1, 2, or 3 and runs the corresponding Lingo if it is:

```
case the rollover of
  1: puppetSound "Horn"
  2: puppetSound "Drum"
  3: puppetSound "Bongos"
end case
```

## castLib

### Syntax

`castLib` *whichCast*

### Description

**Keyword;** indicates that the cast specified in *whichCast* is a cast.

The default cast is cast number 1. To specify a cast member in a cast other than cast 1, set `castLib` to specify the alternative cast.

### Examples

This statement displays the number of the Buttons cast in the Message window.

**Dot syntax:**

```
put castLib("Buttons").number
```

**Verbose syntax:**

```
put the number of castLib "Buttons"
```

This statement assigns cast member 5 in castLib 4 to sprite 10:

```
sprite(10).member = member(5, 4)
```

## castLibNum

### Syntax

```
member(whichCastMember).castLibNum
the castLibNum of member whichCastMember
sprite(whichSprite).castLibNum
the castLibNum of sprite whichSprite
```

### Description

Cast member and sprite property; determines the number of the cast that contains the specified cast member, or which castLib is currently associated with the specified sprite.

If you change the `castLibNum` sprite property without changing the `memberNum` sprite property, Director uses the cast member that has the same cast member number in the new cast. This is useful for movies that you use as templates and update by supplying new casts. If you organize the cast contents so that each cast member has a cast member number that corresponds to its role in the movie, Director automatically inserts the new cast members correctly. To change the cast member assigned to a sprite regardless of its cast, set the `member` sprite property.

For a cast member, this property can be tested but not set. It can be both tested and set for a sprite.

### Examples

This statement determines the number of the cast to which cast member Jazz is assigned.

Dot syntax:

```
put member("Jazz").castLibNum
```

Verbose syntax:

```
put the castLibNum of member "Jazz"
```

The following statement changes the cast member assigned to sprite 5 by switching its cast to Wednesday Schedule.

Dot syntax:

```
sprite(5).castLibNum = castLib("Wednesday Schedule").number
```

Verbose syntax:

```
set the castLibNum of sprite 5 to the number of castLib "Wednesday Schedule"
```

## castMemberList

### Syntax

```
member(whichCursorCastMember).castmemberList  
the castmemberList of member whichCursorCastMember
```

### Description

Cursor cast member property; specifies a list of cast members that make up the frames of a cursor. For *whichCursorCastMember*, substitute a cast member name (within quotation marks) or a cast member number. You can also specify cast members from different casts.

The first cast member in the list is the first frame of the cursor, the second cast member is the second frame, and so on.

If you specify cast members that are invalid for use in a cursor, they will be ignored, and the remaining cast members will be used.

This property can be tested and set.

### Example

This command sets a series of four cast members for the animated color cursor cast member named myCursor.

Dot syntax:

```
member("myCursor").castmemberList = \  
[member 1, member 2, member 1 of castlib 2, member 2 of castlib 2]
```

Verbose syntax:

```
set the castmemberList of member "myCursor" = \  
[member 1, member 2, member 1 of castlib 2, member 2 of castlib 2]
```



## center

### Syntax

`member(whichCastMember).center`  
the center of member *whichCastMember*

### Description

Cast member property; interacts with the `crop` cast member property.

- When the `crop` property is `FALSE`, the `center` property has no effect.
- When `crop` is `TRUE` and `center` is `TRUE`, cropping occurs around the center of the digital video cast member.
- When `crop` is `TRUE` and `center` is `FALSE`, the digital video's right and bottom sides are cropped.

This property can be tested and set.

### Example

This statement causes the digital video cast member `Interview` to be displayed in the top left corner of the sprite.

Dot syntax:

```
member("Interview").center = FALSE
```

Verbose syntax:

```
set the center of member "Interview" to FALSE
```

### See also

`crop` (cast member property), `centerRegPoint`, `regPoint`, `scale`

## centerRegPoint

### Syntax

`member(whichCastMember).centerRegPoint`  
the `centerRegPoint` of member *whichCastMember*

### Description

Flash, vector shape, and bitmap cast member property; automatically centers the registration point of the cast member when you resize the sprite (`TRUE`, default); or repositions the registration point at its current point value when you resize the sprite, set the `defaultRect` property, or set the `regPoint` property (`FALSE`).

This property can be tested and set.

### Example

This script checks to see if a Flash movie's `centerRegPoint` property is set to `TRUE`. If it is, the script uses the `regPoint` property to reposition the sprite's registration point to its upper left corner. By checking the `centerRegPoint` property, the script ensures that it does not reposition a registration point that had been previously set using the `regPoint` property.

### Dot syntax:

```
on beginSprite me
  if sprite(the spriteNum of me).member.centerRegPoint = TRUE then
    sprite(the spriteNum of me).member.regPoint = point(0,0)
  end if
end
```

### Verbose syntax:

```
on beginSprite me
  if the centerRegPoint of member the memberNum of me = TRUE then
    set the regPoint of member the memberNum of me = point(0,0)
  end if
end
```

### See also

regPoint

## centerStage

### Syntax

the centerStage

### Description

Movie property; determines whether the Stage is centered on the monitor when the movie is loaded (TRUE, default) or not centered (FALSE). Place the statement that includes this property in the movie that precedes the movie you want it to affect.

This property is useful for checking the Stage location before a movie plays from a projector.

This property can be tested and set.

**Note:** Be aware that behavior while playing back in a projector differs between Windows and Macintosh systems. Settings selected during creation of the projector may override this property.

### Examples

This statement sends the movie to a specific frame if the Stage is not centered:

```
if the centerStage = FALSE then go to frame "Off Center"
```

This statement changes the centerStage property to the opposite of its current value:

```
set the centerStage to (not the centerStage)
```

### See also

fixStageSize

## changeArea

### Syntax

member(*whichCastMember*).changeArea  
the changeArea of member *whichCastMember*

### Description

Transition cast member property; determines whether a transition applies only to the changing area on the Stage (TRUE) or to the entire Stage (FALSE). Its effect is similar to selecting the Changing Area Only option in the Frame Properties Transition dialog box.

This property can be tested and set.

### Example

This statement makes the transition cast member Wave apply only to the changing area on the Stage.

Dot syntax:

```
member("Wave").changeArea = TRUE
```

Verbose syntax:

```
set the changeArea of member "Wave" to TRUE
```

## channelCount

### Syntax

```
member(whichCastMember).channelCount  
the channelCount of member whichCastMember  
sound(channelNum).channelCount
```

### Description

Sound channel and cast member property; for sound channels, determines the number of channels in the currently playing or paused sound in the given sound channel. For sound cast members, determines the number of channels in the specified cast member.

This is useful for determining whether a sound is in monaural or in stereo. This property can be tested but not set.

### Examples

This statement determines the number of channels in the sound cast member, Jazz.

Dot syntax:

```
put member("Jazz").channelCount
```

Verbose syntax:

```
put the channelCount of member "Jazz"
```

This statement determines the number of channels in the sound member currently playing in sound channel 2:

```
put sound(2).channelCount
```

## char...of

### Syntax

```
textMemberExpression.char[whichCharacter]  
char whichCharacter of fieldOrStringVariable  
textMemberExpression.char[firstCharacter..lastCharacter]  
char firstCharacter to lastCharacter of fieldOrStringVariable
```

### Description

Keyword; identifies a character or a range of characters in a chunk expression. A chunk expression is any character, word, item, or line in any source of text (such as field cast members and variables) that holds a string.

- An expression using *whichCharacter* identifies a specific character.
- An expression using *firstCharacter* and *lastCharacter* identifies a range of characters.

The expressions must be integers that specify a character or range of characters in the chunk. Characters include letters, numbers, punctuation marks, spaces, and control characters such as Tab and Return.

You can test but not set the `char...` of keyword. Use the `put...into` command to modify the characters in a string.

### Examples

This statement displays the first character of the string \$9.00:

```
put (" $9.00").char[1..1]
-- "$"
```

This statement displays the entire string \$9.00:

```
put (" $9.00").char[1..5]
-- "$9.00"
```

This statement changes the first five characters of the second word in the third line of a text cast member:

```
member("quiz").line[3].word[2].char[1..5] = "?????"
```

### See also

`mouseMember`, `mouseItem`, `mouseLine`, `mouseWord`

## characterSet

### Syntax

```
member(whichFontMember).characterSet
the characterSet of member whichFontMember
```

### Description

Font cast member property; returns a string containing the characters included for import when the cast member was created. If all characters in the original font were included, the result is an empty string.

### Example

This statement displays the characters included when cast member 11 was created. The characters included during import were numerals and Roman characters.

```
put member(11).characterSet
-- "1234567890ABCDEFGHIJKLMNQRSTUUVWXYZabcdefghijklmnopqrstuvwxyz"
```

### See also

`recordFont`, `bitmapSizes`, `originalFont`

## charPosToLoc()

### Syntax

```
member(whichCastMember).charPosToLoc(nthCharacter)
charPosToLoc(member whichCastMember, nthCharacter)
```

### Description

Field function; returns the point in the entire field cast member (not just the part that appears on the Stage) that is closest to the character specified by *nthCharacter*. This is useful for determining the location of individual characters.

Values for `charPosToLoc` are in pixels from the top left corner of the field cast member. The *nthCharacter* parameter is 1 for the first character in the field, 2 for the second character, and so on.

#### Example

The following statement determines the point where the fiftieth character in the field cast member *Headline* appears and assigns the result to the variable *location*:

```
location = charPosToLoc(member "Headline", 50)
```

## chars()

#### Syntax

```
chars(stringExpression, firstCharacter, lastCharacter)
```

#### Description

Function; identifies a substring of characters in *stringExpression*. The substring starts at *firstCharacter* and ends at *lastCharacter*. The expressions *firstCharacter* and *lastCharacter* must specify a position in the string.

If *firstCharacter* and *lastCharacter* are equal, then a single character is returned from the string. If *lastCharacter* is greater than the string length, only a substring up to the length of the string is identified. If *lastCharacter* is before *firstCharacter*, the function returns the value `EMPTY`.

To see an example of `chars()` used in a completed movie, see the *Text* movie in the *Learning/Lingo Examples* folder inside the *Director* application folder.

#### Examples

This statement identifies the sixth character in the word *Macromedia*:

```
put chars("Macromedia", 6, 6)
-- "m"
```

This statement identifies the sixth through tenth characters of the word *Macromedia*:

```
put chars("Macromedia", 6, 10)
-- "media"
```

The following statement tries to identify the sixth through twentieth characters of the word *Macromedia*. Because the word has only 10 characters, the result includes only the sixth through tenth characters.

```
put chars("Macromedia", 6, 20)
-- "media"
```

#### See also

`char...of`, `length()`, `offset()` (string function), `number` (characters)

## charSpacing

#### Syntax

```
chunkExpression.charSpacing
```

#### Description

Text cast member property; enables specifying any additional spacing applied to each letter in the *chunkExpression* portion of the text cast member.

A value less than 0 indicates less spacing between letters. A value greater than 0 indicates more spacing between letters.

The default value is 0, which results in default spacing between letters.

### Example

The following handler increases the current character spacing of the third through fifth words within the text cast member myCaption by a value of 2:

```
on myCharSpacer
  mySpaceValue = member("myCaption").word[3..5].charSpacing
  member("myCaption").word[3..5].charSpacing = (mySpaceValue + 2)
end
```

## charToNum()

### Syntax

```
(stringExpression).charToNum
charToNum(stringExpression)
```

### Description

Function; returns the ASCII code that corresponds to the first character of *stringExpression*.

The `charToNum()` function is especially useful for testing the ASCII value of characters created by combining keys, such as the Control key and another alphanumeric key.

Director treats uppercase and lowercase letters the same if you compare them using the equal sign (=) operator; for example, the statement `put ("M" = "m")` returns the result 1 or TRUE.

Avoid problems by using `charToNum()` to return the ASCII code for a character and then use the ASCII code to refer to the character.

### Examples

This statement displays the ASCII code for the letter A:

```
put ("A").charToNum
-- 65
```

The following comparison determines whether the letter entered is a capital A, and then navigates to either a correct sequence or incorrect sequence in the Score:

```
on CheckKeyHit theKey
  if (theKey).charToNum = 65 then
    go "Correct Answer"
  else
    go "Wrong Answer"
  end if
end
```

### See also

`numToChar()`

## checkBoxAccess

### Syntax

the checkBoxAccess

### Description

Movie property; specifies one of three possible results when the user clicks a check box or radio button created with button tools in the Tools window:

- 0 (default)—Lets the user set check boxes and radio buttons on and off.
- 1—Lets the user set check boxes and radio buttons on but not off.
- 2—Prevents the user from setting check boxes and radio buttons at all; the buttons can be set only by scripts.

This property can be tested and set.

### Examples

This statement sets the checkBoxAccess property to 1, which lets the user click check boxes and radio buttons on but not off:

```
the checkBoxAccess to 1
```

This statement records the current setting of the checkBoxAccess property by putting the value in the variable oldAccess:

```
oldAccess to the checkBoxAccess
```

### See also

hilite (cast member property), checkBoxType

## checkBoxType

### Syntax

the checkBoxType

### Description

Movie property; specifies one of three ways to indicate whether a check box is selected:

- 0 (default)—Creates a standard check box that contains an X when the check box is selected.
- 1—Creates a check box that contains a black rectangle when the check box is selected.
- 2—Creates a check box that contains a filled black rectangle when the check box is selected.

This property can be tested and set.

### Example

This statement sets the checkBoxType property to 1, which creates a black rectangle in check boxes when the user clicks them:

```
the checkBoxType to 1
```

### See also

hilite (cast member property), checkBoxAccess

## checkMark

### Syntax

the checkMark of menuItem *whichItem* of menu *whichMenu*

### Description

Menu item property; determines whether a check mark appears next to the custom menu item (TRUE) or not (FALSE, default).

The *whichItem* value can be either a menu item name or a menu item number. The *whichMenu* value can be either a menu name or a menu number.

This property can be tested and set.

**Note:** Menus are not available in Shockwave

### Example

This handler turns off any items that are checked in the custom menu specified by the argument *theMenu*. For example, `unCheck ("Format")` turns off all the items in the Format menu.

```
on unCheck theMenu
  set n = the number of menuItems of menu theMenu
  repeat with i = 1 to n
    set the checkMark of menuItem i of menu theMenu to FALSE
  end repeat
end unCheck
```

### See also

`installMenu`, `enabled`, `name` (menu item property), `number` (menu items), `script`, `menu`

## child

### Syntax

```
member(whichCastmember).model(whichParentNode).\
  child(whichChildNodeName)
member(whichCastmember).model(whichParentNode).child[index]
```

### Description

3D model, group, light, and camera property; returns the child node named *whichChildNodeName* or at the specified index in the parent node's list of children. A node is a model, group, camera, or light.

The transform of a node is parent-relative. If you change the position of the parent, its children move with it, and their positions relative to the parent are maintained. Changes to the rotation and scale properties of the parent are similarly reflected in its children.

Use the `addChild` method of the parent node or set the `parent` property of the child node to add to the parent's list of children. A child can have only one parent, but a parent can have any number of children. A child can also have children of its own.

### Example

This statement shows that the second child of the model named Car is the model named Tire.

```
put member("3D").model("Car").child[2]
-- model("Tire")
```

### See also

`addChild`, `parent`



## child (XML)

### Syntax

`XMLnode.child[ childNumber ]`

### Description

XML property; refers to the specified child node of a parsed XML document's nested tag structure.

### Example

Beginning with the following XML:

```
<?xml version="1.0"?>
  <e1>
    <tagName attr1="val1" attr2="val2"/>
    <e2>element 2</e2>
    <e3>element 3</e3>
    here is some text
  </e1>
```

This Lingo returns the name of the first child node of the preceding XML:

```
put gParserObject.child[1].name
-- "e1"
```

## chunkSize

### Syntax

`member(whichCastMember).chunkSize`  
the *chunkSize* of member *whichCastMember*

### Description

Transition cast member property; determines the transition's chunk size in pixels from 1 to 128 and is equivalent to setting the smoothness slider in the Frame Properties: Transition dialog box. The smaller the chunk size, the smoother the transition appears.

This property can be tested and set.

### Example

This statement sets the chunk size of the transition cast member Fog to 4 pixels.

Dot syntax:

```
member("Fog").chunkSize = 4
```

Verbose syntax:

```
set the chunkSize of member "Fog" to 4
```

## clearAsObjects()

**Help ID:** x5539 | Lingo\_FlashClearAsObject

### Syntax

```
clearAsObjects()
```

### Definition

Command; resets the global Flash Player used for ActionScript objects and removes any ActionScript objects from memory. The command does not clear or reset references to those objects stored in Lingo. Lingo references will persist but will refer to nonexistent objects. You must set each reference to `VOID` individually.

The `clearAsObjects()` command affects only global objects, such as the array created in this statement:

```
myGlobalArray = newObject(#array)
```

The `clearAsObjects()` command has no effect on objects created within sprite references, such as the following:

```
myArray = sprite(2).newObject(#array)
```

### Example

This statement clears all globally created ActionScript objects from memory:

```
clearAsObjects()
```

### See also

`newObject()`, `setCallback()`

## clearAtRender

### Syntax

```
member(whichCastmember).camera(whichCamera).colorBuffer.\
    clearAtRender
sprite(whichSprite).camera{( index )}.colorBuffer.clearAtRender
```

### Description

3D property; indicates whether the color buffer is cleared after each frame. Setting the value to `FALSE`, which means the buffer is not cleared, gives an effect similar to trails ink effect. The default value for this property is `TRUE`.

### Example

This statement prevents Director from erasing past images of the view from the camera. Models in motion will appear to smear across the stage.

```
sprite(1).camera.colorBuffer.clearAtRender = 0
```

### See also

`clearValue`

## clearCache

### Syntax

```
clearCache
```

### Description

Command; clears the Director network cache.

The `clearCache` command clears only the cache, which is separate from the browser's cache.

If a file is in use, it remains in the cache until it is no longer in use.

### Example

This handler clears the cache when the movie starts:

```
on startMovie
    clearCache
end
```

### See also

`cacheDocVerify()`, `cacheSize()`

## clearError

### Syntax

```
member(whichFlashMember).clearError()
clearError (member whichFlashMember)
```

### Description

Flash command; resets the error state of a streaming Flash cast member to 0.

When an error occurs while a cast member is streaming into memory, Director sets the cast member's `state` property to -1 to indicate that an error occurred. When this happens, you can use the `getError` function to determine what type of error occurred and then use the `clearError` command to reset the cast member's error state to 0. After you clear the member's error state, Director tries to open the cast member if it is needed again in the Director movie. Setting a cast member's `pathName`, `linked`, and `preload` properties also automatically clears the error condition.

### Example

This handler checks to see if an out-of-memory error occurred for a Flash cast member named Dali, which was streaming into memory. If a memory error occurred, the script uses the `unloadCast` command to try to free some memory; it then branches the playhead to a frame in the Director movie named Artists, where the Flash movie sprite first appears, so Director can again try to play the Flash movie. If something other than an out-of-memory error occurred, the script goes to a frame named Sorry, which explains that the requested Flash movie can't be played.

```
on CheckFlashStatus
    if member("Dali").getError() = #memory then
        member("Dali").clearError()
        unloadCast
        go to frame "Artists"
    else
        go to frame "Sorry"
    end if
end
```

### See also

`state (Flash, SWA)`, `getError()`

## clearFrame

### Syntax

```
clearFrame
```

### Description

Command; erases everything in the current frame's sprite and affects channels during Score recording only.

**Example**

The following handler clears the content of each frame before it edits that frame during Score generation:

```
on newScore
  beginRecording
  repeat with counter = 1 to 50
    clearFrame
    the frameScript to 25
    updateFrame
  end repeat
endRecording
end
```

**See also**

beginRecording, endRecording, updateFrame

## clearGlobals

**Syntax**

```
clearGlobals
```

**Description**

Command; sets all global variables to VOID.

This command can be useful when initializing global variables or when opening a new movie that requires a new set of global variables.

**Example**

This statement sets all global variables to VOID:

```
clearGlobals
```

## clearValue

**Syntax**

```
member(whichCastmember).camera(whichCamera).colorBuffer\
  .clearValue
sprite(whichSprite).camera{( index )}.colorBuffer.clearValue
```

**Description**

3D property; specifies the color used to clear out the color buffer if `colorBuffer.clearAtRender` is set to TRUE. The default setting for this property is `rgb(0, 0, 0)`.

**Example**

This statement sets the `clearValue` property of the camera to `rgb(255, 0, 0)`. Spaces in the 3d world which are not occupied by models will appear red.

```
sprite(1).camera.colorBuffer.clearValue= rgb(255, 0, 0)
```

**See also**

clearAtRender

## clickLoc

### Syntax

the `clickLoc`

### Description

Function; identifies as a point the last place on the screen where the mouse was clicked.

### Example

The following `on mouseDown` handler displays the last mouse click location:

```
on mouseDown
    put the clickLoc
end mouseDown
```

If the click were 50 pixels from the left end of the Stage and 100 pixels from the top of the Stage, the Message window would display the following:

```
-- point(50, 100)
```

## clickMode

### Syntax

`sprite(whichFlashSprite).clickMode`  
the `clickMode` of sprite *whichFlashSprite*  
`member(whichFlashMember).clickMode`  
the `clickMode` of member *whichFlashMember*

### Description

Flash cast member and sprite property; controls when the Flash movie sprite detects mouse click events (`mouseUp` and `mouseDown`) and when it detects rollovers (`mouseEnter`, `mouseWithin`, and `mouseLeave`). The `clickMode` property can have these values:

- `#boundingBox`—Detects mouse click events anywhere within the sprite's bounding rectangle and detects rollovers at the sprite's boundaries.
- `#opaque` (default)—Detects mouse click events only when the pointer is over an opaque portion of the sprite and detects rollovers at the boundaries of the opaque portions of the sprite if the sprite's ink effect is set to Background Transparent. If the sprite's ink effect is not set to Background Transparent, this setting has the same effect as `#boundingBox`.
- `#object`—Detects mouse click events when the mouse pointer is over any filled (nonbackground) area of the sprite and detects rollovers at the boundaries of any filled area. This setting works regardless of the sprite's ink effect.

This property can be tested and set.

### Example

This script checks to see if the sprite, which is specified with an ink effect of Background Transparent, is currently set to be rendered direct to Stage. If the sprite is not rendered direct to Stage, the sprite's `clickMode` is set to `#opaque`. Otherwise (because ink effects are ignored for Flash movie sprites that are rendered direct to Stage), the sprite's `clickMode` is set to `#boundingBox`.

**Dot syntax:**

```

on beginSprite me
  if sprite(the spriteNum of me).directToStage = FALSE then
    sprite(the spriteNum of me).clickMode = #opaque
  else
    sprite(the spriteNum of me).clickMode = #boundingBox
  end if
end

```

**Verbose syntax:**

```

on beginSprite me
  if the directToStage of sprite the spriteNum of me = FALSE then
    set the clickMode of sprite the spriteNum of me = #opaque
  else
    set the clickMode of sprite the spriteNum of me = #boundingBox
  end if
end

```

## clickOn

**Syntax**

```
the clickOn
```

**Description**

Function; returns the last active sprite clicked by the user. An active sprite is a sprite that has a sprite or cast member script associated with it.

When the user clicks the Stage, `clickOn` returns 0. To detect whether the user clicks a sprite with no script, you must assign a placeholder script to it ("--," for example) so that it can be detected by the `clickOn` function.

The `clickOn` can be checked within a repeat loop. However, neither `clickOn` nor `clickLoc` functions change value when the handler is running. The value that you obtain is the value from before the handler started.

**Examples**

This statement checks whether sprite 7 was the last active sprite clicked:

```
if the clickOn = 7 then alert "Sorry, try again."
```

This statement sets the `foreColor` property of the last active sprite that was clicked to a random color:

```
sprite(the clickOn).foreColor = random(255)-1
```

**See also**

`doubleClick`, the `mouseDown` (system property), `mouseMember`, the `mouseUp` (system property)

# clone

## Syntax

```
member(whichCastmember).model(whichModel).clone(cloneName)  
member(whichCastmember).group(whichGroup).clone(cloneName)  
member(whichCastmember).light(whichLight).clone(cloneName)  
member(whichCastmember).camera(whichCamera).clone(cloneName)
```

## Description

3D command; creates a copy of the model, group, light, or camera and all of its children. The clone is named *cloneName* and shares the parent of the model, group, light, or camera from which it was cloned.

A clone of a model uses the same model resource and is assigned the same shaderList as the original model.

If you do not specify the *cloneName*, or if you specify "", the clone will not be counted by the `count` method, but it will appear in the scene.

## Example

This statement creates a clone named Teapot2 from the model named Teapot, and returns a reference to the new model.

```
teapotCopy = member("3D World").model("Teapot").clone("Teapot2")
```

## See also

```
cloneDeep, cloneModelFromCastmember, cloneMotionFromCastmember, loadFile()
```

# cloneDeep

## Syntax

```
member(whichCastmember).model(whichModel).cloneDeep(cloneName)  
member(whichCastmember).group(whichGroup).cloneDeep(cloneName)  
member(whichCastmember).light(whichLight).cloneDeep(cloneName)  
member(whichCastmember).camera(whichCamera).cloneDeep(cloneName)
```

## Description

3D command; creates a copy of the model, group, light, or camera plus all of the following:

- The model resources, shaders, and textures used by the original model or group
- The children of the model, group, light, or camera
- The model resources, shaders, and textures used by the children

Note that `cloneDeep` uses more memory and takes more time than the `clone` command.

## Example

This statement creates a copy of the model named Teapot, its children, and the model resources, shaders, and textures used by Teapot and its children. The variable `teapotCopy` is a reference to the cloned model.

```
teapotCopy = member("3D World").model("Teapot").cloneDeep("Teapot2")
```

## See also

```
clone, cloneModelFromCastmember, cloneMotionFromCastmember, loadFile()
```

## cloneModelFromCastmember

### Syntax

```
member(whichCastmember).cloneModelFromCastmember\  
    (newModelName, sourceModelName, sourceCastmember)
```

### Description

3D command; copies the model named *sourceModelName* from the cast member *sourceCastmember*, renames it *newModelName*, and inserts it into the cast member *whichCastmember* as a child of its 3D world.

This command also copies the children of *sourceModelName*, as well as the model resources, shaders, and textures used by the model and its children.

The source cast member must be finished loading for this command to work correctly.

### Example

This statement makes a copy of the model named Pluto of the cast member named Scene and inserts it into the cast member named Scene2 with the new name Planet. The children of Pluto are also imported, as are the model resources, shaders, and textures used by Pluto and its children.

```
member("Scene2").cloneModelFromCastmember("Planet", "Pluto", \  
    member("Scene"))
```

### See also

cloneMotionFromCastmember, clone, cloneDeep, loadFile()

## cloneMotionFromCastmember

### Syntax

```
member(whichCastmember).cloneMotionFromCastmember(newMotionName, \  
    sourceMotionName, sourceCastmember)
```

### Description

3D command; copies the motion named *sourceMotionName* from the cast member *sourceCastmember*, renames it *newMotionName*, and inserts it into the cast member *whichCastmember*.

The source cast member must be finished loading for this command to work correctly.

### Example

This statement copies the motion named Walk from the cast member named ParkScene, names the copy FunnyWalk, and puts the copy in the cast member gbMember.

```
member("gbMember").cloneMotionFromCastmember("FunnyWalk", \  
    "Walk", member("ParkScene"))
```

### See also

map (3D), cloneModelFromCastmember, clone, cloneDeep, loadFile()



## closed

### Syntax

`member(whichCastMember).closed`

### Description

Vector shape cast member property; indicates whether the end points of a path are closed or open.

Vector shapes must be closed in order to contain a fill.

The value can be as follows:

- TRUE—the end points are closed.
- FALSE—the end points are open.

## close window

### Syntax

`window(windowIdentifier).close()`  
`close window windowIdentifier`

### Description

Window command; closes the window specified by *windowIdentifier*.

- To specify a window by name, use the syntax `close window name`, where you replace *name* with the name of a window. Use the complete pathname.
- To specify a window by its number in `windowList`, use the syntax `close window number`, where you replace *number* with the window's number in `windowList`.

Closing a window that is already closed has no effect.

Be aware that closing a window does not stop the movie in the window nor clear it from memory. This command simply closes the window in which the movie is playing. You can reopen it quickly by using the `open window` command. This allows rapid access to windows that you want to keep available.

If you want to completely dispose of a window and clear it from memory, use the `forget window` command. Make sure that nothing refers to the movie in that window if you use the `forget window` command, or you will generate errors when scripts try to communicate or interact with the forgotten window.

### Examples

This statement closes the window named `Panel`, which is in the subfolder `MIAW Sources` within the current movie's folder:

```
window("@/MIAW Sources/Panel").close()
```

This statement closes the window that is number 5 in `windowList`:

```
window(5).close()
```

### See also

`forget`, `open window`, `windowList`

## on closeWindow

### Syntax

```
on closeWindow
    statement(s)
end
```

### Description

System message and event handler; contains statements that run when the user closes the window for a movie by clicking the window's close box.

The `on closeWindow` handler is a good place to put Lingo commands that you want executed every time the movie's window closes.

### Example

This handler tells Director to forget the current window when the user closes the window that the movie is playing in:

```
on closeWindow
    -- perform general housekeeping here
    forget the activeWindow
end
```

## closeXlib

### Syntax

```
closeXlib whichFile
```

### Description

Command; closes the Xlibrary file specified by the string *whichFile*. If the Xlibrary file is in a folder other than that for the current movie, *whichFile* must specify a pathname. If no file is specified, all open Xlibraries are closed.

Xtra extensions are stored in Xlibrary files. Xlibrary files are resource files that contain Xtra extensions. HyperCard XCMDs and XFCNs can also be stored in Xlibrary files.

The `closeXlib` command doesn't work for URLs.

In Windows, using the DLL extension for Xtra extensions is optional.

It is good practice to close any file you have opened as soon as you have finished using it.

**Note:** This command is not supported in Shockwave.

### Examples

This statement closes all open Xlibrary files:

```
closeXlib
```

This statement closes the Xlibrary Video Disc Xlibrary when it is in the same folder as the movie:

```
closeXlib "Video Disc Xlibrary"
```

The following statement closes the Xlibrary Transporter Xtra extensions in the folder New Xtras, which is in the same folder as the movie. The disk is identified by the variable *currentDrive*:

```
closeXlib "@:New Xtras:Transporter Xtras"
```

### See also

```
interface(), openXlib, showXlib
```

## collision (modifier)

### Syntax

```
member(whichCastmember).model(whichModel).\  
collision.collisionModifierProperty
```

### Description

3D modifier; manages the detection and resolution of collisions. Adding the `collision` modifier to a model by using the `addModifier` command allows access to the following `collision` modifier properties:

**enabled (collision)** indicates whether collisions with the model are detected.

**resolve** indicates whether collisions with the model are resolved.

**immovable** indicates whether a model can be moved from frame to frame.

**mode (collision)** indicates the geometry used for collision detection.

**Note:** For more detailed information about these properties, see the individual property entries.

The `collision` modifier generates the following events. For more information about using collision events, see the `registerForEvent()` entry.

A `#collideAny` event is generated when a collision occurs between models to which the `collision` modifier has been attached.

A `#collideWith` event is generated when a collision occurs with a specific model to which the `collision` modifier has been attached.

The `collisionData` object is sent as an argument with the `#collideAny` and `#collideWith` events. See the `collisionData` entry for details of its properties.

### See also

`addModifier`, `removeModifier`, `modifiers`

## collisionData

### Syntax

```
on myHandlerName me, collisionData
```

### Description

3D data object; sent as an argument with the `#collideWith` and `#collideAny` events to the handler specified in the `registerForEvent`, `registerScript`, and `setCollisionCallback` commands. The `collisionData` object has these properties:

`modelA` is one of the models involved in the collision.

`modelB` is the other model involved in the collision.

`pointOfContact` is the world position of the collision.

`collisionNormal` is the direction of the collision.

### Example

This example has three parts. The first part is the first line of code, which registers the `#putDetails` handler for the `#collideAny` event. The second part is the `#putDetails` handler. When two models in the cast member `MyScene` collide, the `#putDetails` handler is called and the `collisionData` argument is sent to it. This handler displays the four properties of the `collisionData` object in the message window. The third part of the example shows the results from the message window. The first two lines show that the model named `GreenBall` was model A and the model named `YellowBall` was model B in the collision. The third line shows the point of contact of the two models. The last line shows the direction of the collision.

```
member("MyScene").registerForEvent(#collideAny, #putDetails, 0)

on putDetails me, collisionData
  put collisionData.modelA
  put collisionData.modelB
  put collisionData.pointOfContact
  put collisionData.collisionNormal
end

-- model("GreenBall")
-- model("YellowBall")
-- vector( 24.800, 0.000, 0.000 )
-- vector( -1.000, 0.000, 0.000 )
```

### See also

`collisionData` properties: `modelA`, `modelB`, `pointOfContact`, `collisionNormal`

`collisionData` methods: `resolveA`, `resolveB`, `collision (modifier)`

## collisionNormal

### Syntax

```
collisionData.collisionNormal
```

### Description

3D `collisionData` property; a vector indicating the direction of the collision.

The `collisionData` object is sent as an argument with the `#collideWith` and `#collideAny` events to the handler specified in the `registerForEvent`, `registerScript`, and `setCollisionCallback` commands.

The `#collideWith` and `#collideAny` events are sent when a collision occurs between models to which collision modifiers have been added. The `resolve` property of the models' modifiers must be set to `TRUE`.

This property can be tested but not set.

### Example

This example has two parts. The first part is the first line of code, which registers the `#explode` handler for the `#collideAny` event. The second part is the `#explode` handler. When two models in the cast member named `MyScene` collide, the `#explode` handler is called and the `collisionData` argument is sent to it. The first ten lines of the `#explode` handler create the model resource `SparkSource` and set its properties. This model resource is a single burst of particles. The tenth line sets the direction of the burst to `collisionNormal`, which is the direction of the collision. The eleventh line of the handler creates a model called `SparksModel` using the model resource `SparkSource`. The last line of the handler sets the position of `SparksModel` to the position where the collision occurred. The overall effect is a collision that causes a burst of sparks to fly in the direction of the collision from the point of contact.

```
member("MyScene").registerForEvent(#collideAny, #explode, 0)
on explode me, collisionData
  nmr = member("MyScene").newModelResource("SparkSource", #particle)
  nmr.emitter.mode = #burst
  nmr.emitter.loop = 0
  nmr.emitter.minSpeed = 30
  nmr.emitter.maxSpeed = 50
  nmr.emitter.angle = 45
  nmr.colorRange.start = rgb(0, 0, 255)
  nmr.colorRange.end = rgb(255, 0, 0)
  nmr.lifetime = 5000
  nmr.emitter.direction = collisionData.collisionNormal
  nm = member("MyScene").newModel("SparksModel", nmr)
  nm.transform.position = collisionData.pointOfContact
end
```

### See also

`pointOfContact`, `modelA`, `modelB`, `resolveA`, `resolveB`, `collision` (modifier)

## color()

### Syntax

```
color(#rgb, redValue, greenValue, blueValue)
color(#paletteIndex, paletteIndexNumber)
rgb(rgbHexString)
rgb(redValue, greenValue, blueValue)
paletteIndex(paletteIndexNumber)
```

### Description

Function and data type; determines an object's color as either RGB or 8-bit palette index values. These are the same values as those used in the `color` member and `color` sprite properties, the `bgColor` member and `bgColor` sprite properties, and the `bgColor` Stage property.

The `color` function allows for either 24-bit or 8-bit color values to be manipulated as well as applied to cast members, sprites, and the Stage.

For RGB values, each color component has a range from 0 to 255, and all other values are truncated. For `paletteIndex` types, an integer from 0 to 255 is used to indicate the index number in the current palette, and all other values are truncated.

## Examples

This statement performs a math operation:

```
palColorObj = paletteIndex(20)
put palColorObj
-- paletteIndex(20)
put palColorObj / 2
-- paletteIndex(10)
```

This statement converts one color type to another type:

```
newColorObj = color(#rgb, 155, 0, 75)
put newColorObj
-- rgb(155, 0, 75)
newColorObj.colorType = #paletteIndex
put newColorObj
-- paletteIndex(106)
```

This statement obtains the hexadecimal representation of a color regardless of its type:

```
someColorObj = color(#paletteIndex, 32)
put someColorObj.hexString()
-- "#FF0099"
```

This statement determines individual RGB components and the `paletteIndex` value of a color regardless of its type:

```
newColorObj = color(#rgb, 155, 0, 75)
put newColorObj.green
-- 0
put newColorObj.paletteIndex
-- 106
newColorObj.green = 100
put newColorObj.paletteIndex
-- 94
put newColorObj
-- rgb(155, 100, 75)
newColorObj.paletteIndex = 45
put newColorObj
-- paletteIndex(45)
```

This statement changes the color of the fourth through the seventh characters of text member `myQuotes`:

```
member("myQuotes").char[4..7].color = rgb(200, 150, 75)
```

This Lingo displays the color of sprite 6 in the Message window, and then sets the color of sprite 6 to a new RGB value:

```
put sprite(6).color
-- rgb( 255, 204, 102 )
sprite(6).color = rgb(122, 98, 210)
```

**Note:** Setting the `paletteIndex` value of an RGB color type changes `colorType` to `paletteIndex`. Setting the RGB color type of a `paletteIndex` color sets its `colorType` value to RGB.

## See also

`bgColor`

## color (fog)

### Syntax

```
member(whichCastmember).camera(whichCamera).fog.color  
sprite(whichSprite).camera{(index)}.fog.color
```

### Description

3D property; indicates the color introduced into the scene by the camera when the camera's `fog.enabled` property is set to `TRUE`.

The default setting for this property is `rgb(0, 0, 0)`.

### Example

This statement sets the color of the fog of the camera named `BayView` to `rgb(255, 0, 0)`. If the camera's `fog.enabled` property is set to `TRUE`, models in the fog will take on a red hue.

```
member("MyYard").camera("BayView").fog.color = rgb(255, 0, 0)
```

### See also

`fog`

## color (light)

### Syntax

```
member(whichCastmember).light(whichLight).color
```

### Description

3D light property; indicates the `rgb` value of the light.

The default value of this property is `rgb(191,191,191)`.

### Example

This statement sets the color of the light named `RoomLight` to `rgb(255, 0, 255)`.

```
member("Room").light("RoomLight").color = rgb(255,0,255)
```

### See also

`fog`

## color (sprite and cast member property)

### Syntax

```
sprite(whichSpriteNumber).color  
the color of sprite whichSpriteNumber  
member(whichMember).color
```

### Description

Sprite and text cast member property; for sprites, determines the foreground color of the sprite specified by `whichSprite`. Setting the `foreColor` sprite property is equivalent to choosing the foreground color from the Tools window when the sprite is selected on the Stage.

This property has the equivalent functionality of the `foreColor` sprite property, but the color value returned is a color object of whatever type has been set for that sprite.

For text cast members, this property determines the color of the text.

This property can be tested and set. The color property should be set to an RGB or hexadecimal value.

### Examples

This statement sets the color of the text of cast member 3 to a medium red:

```
member(3).color = rgb(255, 0, 100)
```

This statement sets the color of the text of cast member 3 to a medium blue:

```
member(3).color = rgb("0033FF")
```

### See also

`color()`, `bgColor`, `foreColor`

## colorBufferDepth

### Syntax

```
getRendererServices().colorBufferDepth
```

### Description

3D `rendererServices` property; indicates the color precision of the hardware output buffer of the user's system. The value is either 16 or 32, depending on the user's hardware settings.

This property can be tested but not set.

### Example

This statement shows that the `colorBufferDepth` value of the user's video card is 32.

```
put getRendererServices().colorBufferDepth
-- 32
```

### See also

`getRendererServices()`, `getHardwareInfo()`, `depthBufferDepth`

## colorDepth

### Syntax

```
the colorDepth
```

### Description

System property; determines the color depth of the computer's monitor.

- In Windows, using this property lets you check and set the monitor's color depth. Some video card and driver combinations may not enable you to set the `colorDepth` property. Always verify that the color depth has actually changed after you attempt to set it.
- On the Macintosh, this property lets you check the color depth of different monitors and change it when appropriate.

Possible values are the following:

---

1	Black and white
2	4 colors
4	16 colors
8	256 colors

---



---

16	32,768 or 65,536 colors
32	16,777,216 colors

---

If you try to set a monitor's color depth to a value that monitor does not support, the monitor's color depth doesn't change.

On computers with more than one monitor, the `colorDepth` property refers to the monitor displaying the Stage. If the Stage spans more than one monitor, the `colorDepth` property indicates the greatest depth of those monitors; `colorDepth` tries to set all those monitors to the specified depth.

This property can be tested and set.

### Examples

This statement tells Director to play the segment Full color only if the monitor color depth is set to 256 colors:

```
if the colorDepth = 8 then play movie "Full color"
```

The following handler tries to change the color depth, and if it can't, it displays an alert:

```
on TryToSetColorDepth desiredDepth
  the colorDepth = desiredDepth
  if the colorDepth = desiredDepth then
    return true
  else
    alert "Please change your system to" && desiredDepth &&"color depth and
    reboot."
    return false
  end if
end
```

When changing the user's monitor color depth settings, it is good practice to restore the original depth when the movie has finished. In Windows, the command `set the colorDepth = 0` restores the user's preferred settings from the control panel.

### See also

`switchColorDepth`

## colorList

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
  colorList
member(whichCastmember).modelResource(whichModelResource).\
  colorList[index]
member(whichCastmember).model(whichModel).meshdeform.mesh\
  [meshIndex].colorList
member(whichCastmember).model(whichModel).meshdeform.mesh\
  [meshIndex].colorList[index]
```

### Description

3D property; allows you to get or set every color used in a mesh. This command is accessible only for model resources of the type #mesh. Any single color can be shared by several vertices (faces) of the mesh. Alternately, you can specify texture coordinates for the faces of the mesh and apply a shader to models that use this model resource.

This command must be set to a list of the same number of Lingo color values specified in the `newMesh` call.

### Example

This statement shows that the third color in the `colorList` of the model resource `Mesh2` is `rgb(255, 0, 0)`.

```
put member("shapes").modelResource("mesh2").colorList[3]
-- rgb(255,0,0)
```

### See also

`face`, `colors`

## colorRange

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
    colorRange.start
member(whichCastmember).modelResource(whichModelResource).\
    colorRange.end
```

### Description

3D `#particle` model resource properties; indicate the beginning color and ending color of the particles of a particle system.

The `start` property is the color of the particles when they are created. The `end` property is the color of particles at the end of their lives. The color of each particle gradually changes from the value of `start` to the value of `end` over the course of its life.

The `start` and `end` properties have a default value of `rgb(255, 255, 255)`.

### Example

This statement sets the `colorRange` properties of the model resource named `ThermoSystem`. The first line sets the `start` value to `rgb(255, 0, 0)`, and the second line sets the `end` value to `rgb(0, 0, 255)`. The effect of this statement is that the particles of `ThermoSystem` are red when they first appear, and gradually change to blue during their lifetimes.

```
member(8,2).modelResource("ThermoSystem").colorRange.start = \
    rgb(255,0,0)
member(8,2).modelResource("ThermoSystem").colorRange.end = \
    rgb(0,0,255)
```

### See also

`emitter`, `blendRange`, `sizeRange`

## colors

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
    face[faceIndex].colors
```

### Description

3D `face` property; a linear list of three integers indicating which index positions of the model resource's color list to use for the three vertices of the face. The color list is a linear list of `rgb` values.

The `colors` property is used only with model resources whose type is `#mesh`.

You must use the model resource's `build()` command after setting this property; otherwise, the changes will not take effect.

### Example

This example creates a model resource whose type is `#mesh`, specifies its properties, and then creates a new model with it.

Line 1 uses the `newMesh()` command to create a `#mesh` model resource named `Triangle`, which has one face, three vertices, and a maximum of three colors. The number of normals and the number of texture coordinates are not set.

Line 2 sets the `vertexList` property to a list of three vectors.

Line 3 assigns the vectors of the `vertexList` property to the vertices of the first face of `Triangle`.

Line 4 sets the `colorList` to three `rgb` values.

Line 5 assigns colors to the first face of `Triangle`. The third color in the `colorList` is applied to the first vertex of `Triangle`, the second color to the second vertex, and the first color to the third vertex. The colors will spread across the first face of `Triangle` in gradients.

Line 6 creates the normals of `Triangle` with the `generateNormals()` command.

Line 7 uses the `build()` command to construct the mesh.

Line 8 creates a new model named `TriModel` that uses the new mesh.

```
nm = member("Shapes").newMesh("Triangle",1,3,0,3,0)
nm.vertexList = [vector(0,0,0), vector(20,0,0), vector(20, 20, 0)]
nm.face[1].vertices = [1,2,3]
nm.colorList = [rgb(255,255,0), rgb(0, 255, 0), rgb(0,0,255)]
nm.face[1].colors = [3,2,1]
nm.generateNormals(#smooth)
nm.build()
nm = member("Shapes").newModel("TriModel", nm)
```

### See also

`face`, `vertices`, `colorList`, `flat`

## colorSteps

### Syntax

```
member(whichCastmember).model(whichModel).toon.colorSteps
member(whichCastmember).model(whichModel).shader.colorSteps
member(whichCastmember).shader(whichShader).colorSteps
```

### Description

3D `toon` modifier and `painter` shader property; the maximum number of colors available for use by the `toon` modifier or `painter` shader. The value of this property can be 2, 4, 8, or 16. If you set the value of `colorSteps` to any other number, it will be rounded to one of these.

The default value is 2.

### Example

This statement limits the number of colors available for use by the `toon` modifier for the model named `Teapot` to 8. The teapot will be rendered with a maximum of eight colors.

```
member("shapes").model("Teapot").toon.colorSteps = 8
```

**See also**

highlightPercentage, shadowPercentage

# commandDown

## Syntax

the `commandDown`

## Description

Function; determines whether the Control key (Windows) or the Command key (Macintosh) is being pressed (TRUE) or not (FALSE).

You can use `commandDown` together with the element `the key` to determine when the Control or Command key is pressed in combination with another key. This lets you create handlers that are executed when the user presses specified Control or Command key combinations.

Control or Command key equivalents for the Director authoring menus take precedence while the movie is playing, unless you have installed custom Lingo menus or are playing a projector version of the movie.

For a movie playing back with the Director player for Java, this function returns TRUE only if a second key is pressed simultaneously with the Control or Command key. If the Control or Command key is pressed by itself, `commandDown` returns FALSE. This is because the browser receives the keys before the movie and thus responds to and intercepts any key combinations that are also browser keyboard shortcuts. For example, if the user presses Control+R or Command+R, the browser reloads the current page; the movie never receives the key combination.

## Example

These statements pause a projector whenever the user presses Control+A or Command+A. By setting the `keyDownScript` property to `doCommandKey`, the `on prepareMovie` handler makes the `doCommandKey` handler the first event handler executed when a key is pressed. The `doCommandKey` handler checks whether the Control+A or Command+A keys are pressed at the same time and pauses the movie if they are.

```
on prepareMovie
    the keyDownScript = "doCommandKey"
end

on doCommandKey
    if (the commandDown) and (the key = "a") then go to the frame
end
```

## See also

`controlDown`, `key()`, `keyCode()`, `optionDown`, `shiftDown`

# comments

## Syntax

*member*.`comments`  
the `comments` of *member*

## Description

This cast member property provides a place to store any comments you want to maintain about the given cast member, or any other strings you want to associate with the member. This property can be tested and set. It can also be set in the Property inspector's Member tab.

### Example

This statement sets the comments of the member Backdrop to the string “Still need to license this artwork”:

```
member("Backdrop").comments = "Still need to license this artwork"
```

### See also

`creationDate`, `modifiedBy`, `modifiedDate`

## compressed

### Syntax

```
member(whichCastmember).texture(whichTexture).compressed
```

### Description

3D texture property; indicates whether the source cast member of the texture is compressed (TRUE) or not (FALSE). The value of the `compressed` property changes automatically from TRUE to FALSE when the texture is needed for rendering. It can be set to FALSE to decompress the texture at an earlier time. It can be set to TRUE to release the decompressed representation from memory. Cast members used for textures will not be compressed if this value is TRUE (apart from the standard compression used for bitmap cast members when a Director movie is saved). The default value for this property is TRUE.

### Example

This statement sets the `compressed` property of the texture Plutomap to TRUE.

```
member("scene").texture("Plutomap").compressed = TRUE
```

### See also

`texture`

## constrainH()

### Syntax

```
constrainH (whichSprite, integerExpression)
```

### Description

Function; evaluates *integerExpression* and then returns a value that depends on the horizontal coordinates of the left and right edges of *whichSprite*, as follows:

- When the value is between the left and right coordinates, the value doesn't change.
- When the value is less than the left horizontal coordinate, the value changes to the value of the left coordinate.
- When the value is greater than the right horizontal coordinate, the value changes to the value of the right coordinate.

The `constrainH` and `constrainV` functions constrain only one axis each; the `constrain` sprite property limits both. Note that this function does not change the sprite's properties.

### Examples

These statements check the `constrainH` function for sprite 1 when it has left and right coordinates of 40 and 60:

```
put constrainH(1, 20)
-- 40

put constrainH(1, 55)
-- 55

put constrainH(1, 100)
-- 60
```

This statement constrains a moveable slider (sprite 1) to the edges of a gauge (sprite 2) when the mouse pointer goes past the edge of the gauge:

```
set the locH of sprite 1 to constrainH(2, the mouseH)
```

### See also

`constrainV()`, `constraint`, `left`, `right`

## constraint

### Syntax

`sprite(whichSprite).constraint`  
the constraint of sprite *whichSprite*

### Description

Sprite property; determines whether the registration point of the sprite specified by *whichSprite* is constrained to the bounding rectangle of another sprite (1 or `TRUE`) or not (0 or `FALSE`, default).

The `constraint` sprite property is useful for constraining a moveable sprite to the bounding rectangle of another sprite to simulate a track for a slider control or to restrict where on the screen a user can drag an object in a game.

The `constraint` sprite property affects moveable sprites and the `locH` and `locV` sprite properties. The constraint point of a moveable sprite cannot be moved outside the bounding rectangle of the constraining sprite. (The constraint point for a bitmap sprite is the registration point. The constraint point for a shape sprite is its top left corner.) When a sprite has a constraint set, the constraint limits override any `locH` and `locV` sprite property settings.

This property can be tested and set.

### Examples

This statement removes a `constraint` sprite property:

Dot syntax:

```
sprite(whichSprite).constraint = 0
```

Verbose syntax:

```
set the constraint of sprite whichSprite to 0
```

This statement constrains sprite (i + 1) to the boundary of sprite 14:

```
sprite(i + 1).constraint = 14
```

This statement checks whether sprite 3 is constrained and activates the handler `showConstraint` if it is (the operator `<>` performs a not-equal-to operation):

```
if sprite(3).constraint <> 0 then showConstraint
```

**See also**

`constrainH()`, `constrainV()`, `locH`, `locV`

## constrainV()

**Syntax**

`constrainV (whichSprite, integerExpression)`

**Description**

Function; evaluates *integerExpression* and then returns a value that depends on the vertical coordinates of the top and bottom edges of the sprite specified by *whichSprite*, as follows:

- When the value is between the top and bottom coordinates, the value doesn't change.
- When the value is less than the top coordinate, the value changes to the value of the top coordinate.
- When the value is greater than the bottom coordinate, the value changes to the value of the bottom coordinate.

This function does not change the sprite properties.

**Examples**

These statements check the `constrainV` function for sprite 1 when it has top and bottom coordinates of 40 and 60:

```
put constrainV(1, 20)
-- 40

put constrainV(1, 55)
-- 55

put constrainV(1, 100)
-- 60
```

This statement constrains a moveable slider (sprite 1) to the edges of a gauge (sprite 2) when the mouse pointer moves past the edge of the gauge:

```
set the locV of sprite 1 to constrainV(2, the mouseV)
```

**See also**

`bottom`, `constraint`, `top`, `constrainH()`

## contains

**Syntax**

*stringExpression1* contains *stringExpression2*

**Description**

Operator; compares two strings and determines whether *stringExpression1* contains *stringExpression2* (TRUE) or not (FALSE).

The `contains` comparison operator has a precedence level of 1.

The `contains` comparison operator is useful for checking whether the user types a specific character or string of characters. You can also use the `contains` operator to search one or more fields for specific strings of characters.



### Example

This example determines whether a character passed to it is a digit:

```
on isNumber aLetter
  digits = "1234567890"
  if digits contains aLetter then
    return TRUE
  else
    return FALSE
  end if
end
```

**Note:** The string comparison is not sensitive to case or diacritical marks; “a” and Å are treated the same.

### See also

`offset()` (string function), `starts`

## continue

This Lingo is obsolete. Use `go to the frame +1`.

## controlDown

### Syntax

`the controlDown`

### Description

Function; determines whether the Control key is being pressed (TRUE) or not (FALSE).

You can use the `controlDown` function together with the `key` to check for combinations of the Control key and another key.

For a movie playing back with the Director player for Java, this function returns TRUE only if a second key is pressed simultaneously with the Control key. If the Control key is pressed by itself, `controlDown` returns FALSE. The Director player for Java supports key combinations with the Control key. However, the browser receives the keys before the movie and thus responds to and intercepts any key combinations that are also browser keyboard shortcuts.

For a demonstration of modifier keys and Lingo, see the sample movie *Keyboard Lingo* in Director Help.

### Example

This `on keyDown` handler checks whether the pressed key is the Control key, and if it is, the handler activates the `on doControlKey` handler. The argument (`the key`) identifies which key was pressed in addition to the Control key.

```
on keyDown
  if (the controlDown) then doControlKey (the key)
end
```

### See also

`charToNum()`, `commandDown`, `key()`, `keyCode()`, `optionDown`, `shiftDown`

## controller

### Syntax

`member(whichCastMember).controller`  
the controller of member *whichCastMember*

### Description

Digital video cast member property; determines whether a digital video movie cast member shows or hides its controller. Setting this property to 1 shows the controller; setting it to 0 hides the controller.

The `controller` member property applies to a QuickTime digital video only.

- Setting the `controller` member property for a Video for Windows digital video performs no operation and generates no error message.
- Checking the `controller` member property for a Video for Windows digital video always returns `FALSE`.

The digital video must be in direct-to-stage playback mode to display the controller.

### Example

This statement causes the QuickTime cast member `Demo` to display its controller.

Dot syntax:

```
member("Demo").controller = 1
```

Verbose syntax:

```
set the controller of member "Demo" to 1
```

### See also

`directToStage`

## copyPixels()

### Syntax

```
imageObject.copyPixels(sourceImageObject, destinationRect, sourceRect  
    {, parameterList})  
imageObject.copyPixels(sourceImageObject, destinationQuad, sourceRect  
    {, parameterList })
```

### Description

This function copies the contents of the *sourceRect* from the *sourceImageObject* into the given *imageObject*. The pixels are copied from the *sourceRect* in the *sourceImageObject* and placed into the *destinationRect* or *destinationQuad* in the given *imageObject*. See `quad` for information on using quads.

You can include an optional property list of parameters in order to manipulate the pixels being copied before they are placed into the *destinationRect*. The property list may contain any or all of the following parameters:

Property	Use and Effect
#color	The foreground color to apply for colorization effects. The default color is black.
#bgColor	The background color to apply for colorization effects or background transparency. The default background color is white.
#ink	The type of ink to apply to the copied pixels. This can be an ink symbol or the corresponding numeric ink value. The default ink is #copy. See ink for the list of possible values.
#blendLevel	The degree of blend (transparency) to apply to the copied pixels. The range of values is from 0 to 255. The default value is 255 (opaque). Using a value less than 255 forces the #ink setting to be #blend, or #blendTransparent if it was originally #backgroundTransparent. You may also substitute #blend as the property name and use a value range of 0 to 100. See blend for more information.
#dither	A TRUE or FALSE value that determines whether the copied pixels will be dithered when placed into the <i>destinationRect</i> in 8- and 16-bit images. The default value is FALSE, which maps the copied pixels directly into the <i>imageObject</i> 's color palette.
#useFastQuads	A TRUE or FALSE value that determines whether quad calculations are made using the faster but less precise method available in Director when copying pixels into a <i>destinationQuad</i> . Set this to TRUE if you are using quads for simple rotation and skew operations. Leave it FALSE (the default value) for arbitrary quads, such as those used for perspective transformations. See useFastQuads.
#maskImage	Used to specified a mask or matte object created with the <code>createMask()</code> or <code>createMatte()</code> functions to be used as a mask for the pixels being copied. This allows you to duplicate the effects of mask and matte sprite inks. Note that if the source image has an alpha channel and its <code>useAlpha</code> property is TRUE, the alpha channel is used and the specified mask or matte is ignored. The default is no mask.
#maskOffset	A point indicating the amount of x and y offset to apply to the mask specified by #maskImage. The offset is relative to the upper left corner of the <i>sourceImage</i> . The default offset is (0, 0).

When copying pixels from one area of a cast member to another area of the same member, it is best to copy the pixels first into a duplicate image object before copying them back into the original member. Copying directly from one area to another in the same image is not recommended. See `duplicate()`.

To simulate matte ink with `copyPixels()`, create a matte object with `createMatte()` and then pass that object as the #maskImage parameter with `copyPixels()`.

Copying pixels from an image object into itself is not recommended. Use separate image objects instead.

To see an example of `quad` used in a completed movie, see the Quad movie in the Learning/Lingo Examples folder inside the Director application folder.

### Examples

This statement copies the entire image of member Happy into the rectangle of member flower. If the members are different sizes, the image of member Happy will be resized to fit the rectangle of member flower.

```
member("flower").image.copyPixels(member("Happy").image, \
member("flower").rect, member("Happy").rect)
```

The following statement copies part of the image of member Happy into part of member flower. The part of the image copied from Happy is within rectangle(0, 0, 200, 90). It is pasted into rectangle(20, 20, 100, 40) within the image of member flower. The copied portion of Happy is resized to fit the rectangle into which it is pasted.

```
member("flower").image.copyPixels(member("Happy").image,\nrect(20, 20, 100, 40), rect(0, 0, 200, 90))
```

The following statement copies the entire image of member Happy into a rectangle within the image of member flower. The rectangle into which the copied image of member Happy is pasted is the same size as the rectangle of member Happy, so the copied image is not resized. The blend level of the copied image is 50, so it is semi-transparent, revealing the part of member flower it is pasted over.

```
member("flower").image.copyPixels(member("Happy").image,\nmember("Happy").rect, member("Happy").rect, [#blendLevel: 50])
```

#### See also

ink, color()

## copyrightInfo

### Syntax

```
member(whichCastMember).copyrightInfo  
copyrightInfo of member whichCastMember
```

### Description

Shockwave Audio (SWA) cast member property; displays the copyright text in a SWA file. This property is available only after the SWA sound begins playing or after the file has been preloaded using the `preLoadBuffer` command.

This property can be tested and set.

### Example

This statement tells Director to display the copyright information for the Shockwave Audio file SWAfile in a field cast member named Info Display.

#### Dot syntax:

```
set whatState = the state of member "SWAfile"  
if whatState > 1 AND whatState < 9 then  
    put member("SWAfile").copyrightInfo into member("Info Display")  
end if
```

#### Verbose syntax:

```
set whatState = the state of member "SWAfile"  
if whatState > 1 AND whatState < 9 then  
    put the copyrightInfo of member "SWAfile" into member "Info Display"  
end if
```

# copyToClipboard

## Syntax

```
member(whichCastMember).copyToClipboard()  
copyToClipboard member whichCastMember
```

## Description

Command; copies the specified cast member to the Clipboard without requiring that the cast window is active. You can use this command to copy cast members between movies or applications.

## Examples

This statement copies the cast member named chair to the Clipboard:

```
member("chair").copyToClipboard()
```

This statement copies cast member number 5 to the Clipboard:

```
member(5).copyToClipboard()
```

## See also

pasteClipboardInto

# cos()

## Syntax

```
(angle).cos  
cos (angle)
```

## Description

Function; calculates the cosine of the specified angle, which must be expressed in radians.

## Example

The following statement calculates the cosine of PI divided by 2 and displays it in the Message window:

```
put (PI/2).cos
```

## See also

atan(), PI, sin()

# count()

## Syntax

```
list.count  
count (list)  
count(theObject)  
object.count  
textExpression.count
```

## Description

Function; returns the number of entries in a linear or property list, the number of properties in a parent script without counting the properties in an ancestor script, or the chunks of a text expression such as characters, lines, or words.

The count command works with linear and property lists, objects created with parent scripts, and the globals property.

To see an example of `count()` used in a completed movie, see the Text movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This statement displays the number 3, the number of entries:

```
put [10,20,30].count
-- 3
```

### See also

`globals`

## count

### Syntax

```
member(whichCastmember).light.count
member(whichCastmember).camera.count
member(whichCastmember).modelResource(whichModelResource).\
    bone.count
member(whichCastmember).model.count
member(whichCastmember).group.count
member(whichCastmember).shader.count
member(whichCastmember).texture.count
member(whichCastmember).modelResource.count
member(whichCastmember).motion.count
member(whichCastmember).light.child.count
member(whichCastmember).camera.child.count
member(whichCastmember).model.child.count
member(whichCastmember).group.child.count
sprite(whichSprite).camera{( index )}.backdrop.count
member(whichCastmember).camera(whichCamera).backdrop.count
sprite(whichSprite).camera{( index )}.overlay.count
member(whichCastmember).camera(whichCamera).overlay.count
member(whichCastmember).model(whichModel).modifier.count
member(whichCastmember).model(whichModel).keyframePlayer.\
    playlist.count
member(whichCastmember).model(whichModel).bonesPlayer.\
    playlist.count
member(whichCastmember).modelResource(whichModelResource).\
    face.count
member(whichCastmember).model(whichModel).meshDeform.\
    mesh[ index ].textureLayer.count
member(whichCastmember).model(whichModel).meshDeform.mesh.count
member(whichCastmember).model(whichModel).meshDeform.\
    mesh[ index ].face.count
```

### Description

3D property; returns the number of items in the given list that is associated with the given 3D object. Can be used with any type of object.

The `face.count` property allows you to get the number of triangles in the mesh for a model resource whose type is `#mesh`.

This property can be tested but not set.

### Examples

These examples determine the number of various types of objects within a 3D cast member called 3D World.

```
numberOfCameras = member("3D World").camera.count
put member("3D World").light.count
-- 3
numberOfModels = member("3D World").model.count
numberOfTextures = member("3D World").texture.count
put member("3D World").modelResource("mesh2").face.count
-- 4
```

This statement shows that the first mesh of the model named Ear is composed of 58 faces.

```
put member("Scene").model("Ear").meshdeform.mesh[1].face.count
-- 58
```

This statement shows that the model named Ear is composed of three meshes.

```
put member("Scene").model("Ear").meshdeform.mesh.count
-- 3
```

This statement shows that the first mesh of the model named Ear has two texture layers.

```
put member("Scene").model("Ear").meshdeform.mesh[1].\
    textureLayer.count
-- 2
```

### See also

cameraCount()

## cpuHogTicks

### Syntax

the cpuHogTicks

### Description

System property; determines how often Director releases control of the CPU to let the computer process background events, such as events in other applications, network events, clock updates, and other keyboard events.

The default value is 20 ticks. To give more time to Director before releasing the CPU to background events or to control how the computer responds to network operations, set `cpuHogTicks` to a higher value.

To create faster auto-repeating key performance but slower animation, set `cpuHogTicks` to a lower value. In a movie, when a user holds down a key to generate a rapid sequence of auto-repeating key presses, Director typically checks for auto-repeating key presses less frequently than the rate set in the computer's control panel.

The `cpuHogTicks` property works only on the Macintosh.

### Example

This statement tells Director to release control of the CPU every 6 ticks, or every 0.10 of a second:

```
the cpuHogTicks = 6
```

### See also

ticks

## creaseAngle

### Syntax

```
member(whichCastmember).model(whichModel).inker.creaseAngle  
member(whichCastmember).model(whichModel).toon.creaseAngle
```

### Description

3D `inker` and `toon` modifier property; indicates the sensitivity of the line drawing function of the modifier to the presence of creases in the model's geometry. Higher settings result in more lines (detail) drawn at creases.

The `creases` property of the modifier must be set to `TRUE` for the `creaseAngle` property to have an effect.

`CreaseAngle` has a range of -1.0 to +1.0. The default setting is 0.01.

### Example

This statement sets the `creaseAngle` property of the `inker` modifier applied to the model named `Teapot` to 0.10. A line will be drawn at all creases in the model that exceed this threshold. This setting will only take effect if the `inker` modifier's `creases` property is set to `TRUE`.

```
member("shapes").model("Teapot").inker.creaseAngle = 0.10
```

### See also

`creases`, `lineColor`, `lineOffset`, `useLineOffset`

## creases

### Syntax

```
member(whichCastmember).model(whichModel).inker.creases  
member(whichCastmember).model(whichModel).toon.creases
```

### Description

3D `toon` and `inker` modifier property; determines whether lines are drawn at creases in the surface of the model.

The default setting for this property is `TRUE`.

### Example

This statement sets the `creases` property of the `inker` modifier for the model named `Teapot` to `TRUE`. A line will be drawn on all creases in the model that exceed the threshold set by the `inker` modifier's `creaseAngle` property.

```
member("shapes").model("Teapot").inker.creases = TRUE
```

### See also

`creaseAngle`, `lineColor`, `lineOffset`, `useLineOffset`



## createMask()

### Syntax

```
imageObject.createMask()
```

### Description

This function creates and returns a mask object for use with the `copyPixels()` function.

Mask objects aren't image objects; they're useful only with the `copyPixels()` function for duplicating the effect of mask sprite ink. To save time, if you plan to use the same image as a mask more than once, it's best to create the mask object and save it in a variable for reuse.

### Example

This statement copies the entire image of member Happy into a rectangle within the image of member brown square. Member gradient2 is used as a mask with the copied image. The mask is offset by 10 pixels up and to the left of the rectangle into which the image of member Happy is pasted.

```
member("brown square").image.copyPixels(member("Happy").image, \
rect(20, 20, 150, 108), member("Happy").rect, \
[#maskImage:member("gradient2").image.createMask(), maskOffset:point(-10, -
10)])
```

### See also

`copyPixels()`, `createMatte()`, `ink`

## createMatte()

### Syntax

```
imageObject.createMatte({alphaThreshold})
```

### Description

This function creates and returns a matte object that you can use with `copyPixels()` to duplicate the effect of the matte sprite ink. The matte object is created from the specified image object's alpha layer. The optional parameter *alphaThreshold* excludes from the matte all pixels whose alpha channel value is below that threshold. It is used only with 32-bit images that have an alpha channel. The *alphaThreshold* must be a value between 0 and 255.

Matte objects aren't image objects; they are useful only with the `copyPixels()` function. To save time, if you plan to use the same image as a matte more than once, it's best to create the matte and save it in a variable for reuse.

### Example

This statement creates a new matte object from the alpha layer of the image object testImage and ignores pixels with alpha values below 50%:

```
newMatte = testImage.createMatte(128)
```

### See also

`copyPixels()`, `createMask()`

## creationDate

### Syntax

*member.creationDate*  
the *creationDate* of *member*

### Description

This cast member property records the date and time that the cast member was first created, using the system time on the computer. You can use this property to schedule a project; Director doesn't use it for anything.

This property can be tested and set.

### Example

Although you typically inspect the *creationDate* property using the Property inspector or the Cast window list view, you can check it in the Message window:

```
put member(1).creationDate
-- date( 1999, 12, 8 )
```

### See also

*comments*, *modifiedBy*, *modifiedDate*

## crop() (image object command)

### Syntax

*imageObject.crop(rectToCropTo)*

### Description

When used with an image object, returns a new image object that contains a copy of the given image object, cropped to the given rect. The original image object is unchanged. The new image object does not belong to any cast member and has no association with the Stage. If you wish to assign it to a cast member you can do so by setting the *image* property of that cast member.

This is different from using *crop* with a cast member, which crops the cast member itself, altering the original.

### Examples

This Lingo takes a snapshot of the Stage and crops it to the *rect* of sprite 10, capturing the current appearance of that sprite on the Stage:

```
set stageImage = (the stage).image
set spriteImage = stageImage.crop(sprite(10).rect)
member("sprite snapshot").image = spriteImage
```

This statement uses the rectangle of cast member Happy to crop the image of cast member Flower, then sets the image of cast member Happy to the result:

```
member("Happy").image = member("Flower").image.crop(member("Happy").rect)
```

## crop() (member command)

### Syntax

```
member(whichMember).crop(rectToCropTo)  
crop member whichMember, rectToCropTo
```

### Description

Bitmap command; allows a bitmap cast member to be cropped to a specific size.

You can use `crop` to trim existing cast members, or in conjunction with the picture of the Stage to grab a snapshot and then crop it to size for display.

The registration point is kept in the same location so the bitmap does not move in relation to the original position.

### Example

This statement sets an existing bitmap member to a snapshot of the Stage, then crops the resulting image to a rectangle equal to sprite 10:

```
member("stage image").picture = (the stage).picture  
member("stage image").crop(sprite(10).rect)
```

### See also

`picture` (cast member property)

## crop (cast member property)

### Syntax

```
member(whichCastMember).crop  
the crop of member whichCastMember
```

### Description

Cast member property; scales a digital video cast member to fit exactly inside the sprite rectangle in which it appears (FALSE), or it crops but doesn't scale the cast member to fit inside the sprite rectangle (TRUE).

This property can be tested and set.

### Example

This statement instructs Lingo to crop any sprite that refers to the digital video cast member Interview.

Dot syntax:

```
member("Interview").crop = TRUE
```

Verbose syntax:

```
set the crop of member "Interview" to TRUE
```

### See also

`center`

## cross

### Syntax

```
vector1.cross(vector2)
```

### Description

3D vector method; returns a vector which is perpendicular to both *vector1* and *vector2*.

### Example

In this example, *pos1* is a vector on the x axis and *pos2* is a vector on the y axis. The value returned by *pos1.cross(pos2)* is *vector( 0.0000, 0.0000, 1.00000e4 )*, which is perpendicular to both *pos1* and *pos2*.

```
pos1 = vector(100, 0, 0)
pos2 = vector(0, 100, 0)
put pos1.cross(pos2)
-- vector( 0.0000, 0.0000, 1.00000e4 )
```

### See also

*crossProduct()*, *perpendicularTo*

## crossProduct()

### Syntax

```
vector1.crossProduct(vector2)
```

### Description

3D vector method; returns a vector which is perpendicular to both *vector1* and *vector2*.

### Example

In this example, *pos1* is a vector on the x axis and *pos2* is a vector on the y axis. The value returned by *pos1.crossProduct(pos2)* is *vector( 0.0000, 0.0000, 1.00000e4 )*, which is perpendicular to both *pos1* and *pos2*.

```
pos1 = vector(100, 0, 0)
pos2 = vector(0, 100, 0)
put pos1.crossProduct(pos2)
-- vector( 0.0000, 0.0000, 1.00000e4 )
```

### See also

*perpendicularTo*, *cross*

## on cuePassed

### Syntax

```
on cuePassed(channelID, cuePointNumber, cuePointName)
    statement(s)
end
on cuePassed(me, channelID, cuePointNumber, cuePointName)
    statement(s)
end
```

## Description

System message and event handler; contains statements that run each time a sound or sprite passes a cue point in its media.

- *me*—The optional *me* parameter is the `scriptInstanceRef` value of the script being invoked. You must include this parameter when using the message in a behavior. If this parameter is omitted, the other arguments will not be processed correctly.
- *channelID*—The number of the sound or sprite channel for the file where the cue point occurred.
- *cuePointNumber*—The ordinal number of the cue point that triggers the event in the list of the cast member's cue points.
- *cuePointName*—The name of the cue point that was encountered.

The message is passed—in order—to sprite, cast member, frame, and movie scripts. For the sprite to receive the event, it must be the source of the sound, like a QuickTime movie or SWA cast member. Use the `isPastCuePoint` property to check cues in behaviors on sprites that don't generate sounds.

## Example

This handler placed in a Movie or Frame script reports any cue points in sound channel 1 to the Message window:

```
on cuePassed channel, number, name
  if (channel = #Sound1) then
    put "CuePoint" && number && "named" && name && "occurred in sound 1"
  end if
end
```

## See also

`scriptInstanceList`, `cuePointNames`, `cuePointTimes`, `isPastCuePoint()`

# cuePointNames

## Syntax

`member(whichCastMember).cuePointNames`  
the `cuePointNames` of member *whichCastMember*

## Description

Cast member property; creates list of cue point names, or if a cue point is not named, inserts an empty string (" ") as a placeholder in the list. Cue point names are useful for synchronizing sound, QuickTime, and animation.

This property is supported by SoundEdit cast members, QuickTime digital video cast members, and Xtra extension cast members that contain cue points. Xtra extensions that generate cue points at run time may not be able to list cue point names.

## Example

This statement obtains the name of the third cue point of a cast member.

Dot syntax:

```
put member("symphony").cuePointNames[3]
```

Verbose syntax:

```
put (getAt(the cuePointNames of member "symphony",3))
```

### See also

`cuePointTimes`, `mostRecentCuePoint`

## cuePointTimes

### Syntax

```
member(whichCastMember).cuePointTimes  
the cuePointTimes of member whichCastMember
```

### Description

Cast member property; lists the times of the cue points, in milliseconds, for a given cast member. Cue point times are useful for synchronizing sound, QuickTime, and animation.

This property is supported by SoundEdit cast members, QuickTime digital video cast members, and Xtra extension cast members that support cue points. Xtra extensions that generate cue points at run time may not be able to list cue point names.

### Example

This statement obtains the time of the third cue point for a sound cast member.

Dot syntax:

```
put member("symphony").cuePointTimes[3]
```

Verbose syntax:

```
put (getAt(the cuePointTimes of member "symphony",3))
```

### See also

`cuePointNames`, `mostRecentCuePoint`

## currentLoopState

### Syntax

```
member(whichCastmember).model(whichModel).keyframePlayer.\  
    currentLoopState  
member(whichCastmember).model(whichModel).bonesPlayer.\  
    currentLoopState
```

### Description

3D `keyframePlayer` and `bonesPlayer` modifier property; indicates whether the motion being executed by the model repeats continuously (TRUE) or plays to the end and is replaced by the next motion in the modifier's playlist (FALSE).

The default setting for this property is the value of the looped parameter of the `play()` command that initiated playback of the motion, or the value of the `queue()` command that added the motion to the modifier's playlist. Changing the `currentLoopState` property also changes the value of the `#looped` property of the motion's entry in the modifier's playlist.

### Example

This statement causes the motion that is being executed by the model named Monster to repeat continuously.

```
member("NewAlien").model("Monster").keyframePlayer.\  
    currentLoopState = TRUE
```

### See also

`loop` (cast member property), `play()` (3D), `queue()` (3D), `playlist`

## currentSpriteNum

### Syntax

the `currentSpriteNum`

### Description

Movie property; indicates the channel number of the sprite whose script is currently running. It is valid in behaviors and cast member scripts. When used in frame scripts or movie scripts, the `currentSpriteNum` property's value is 0.

The `currentSpriteNum` property is similar to `spriteNum` of `me`, but it doesn't require the `me` reference.

This property can be tested but not set.

**Note:** This property was more useful during transitions from older movies to Director 6, when behaviors were introduced. It allowed some behavior-like functionality without having to completely rewrite Lingo code. It is not necessary when authoring with behaviors and is therefore less useful than in the past.

### Example

The following handler in a cast member or movie script switches the cast member assigned to the sprite involved in the `mouseDown` event:

```
on mouseDown
    sprite(the currentSpriteNum).member = member "DownPict"
end
```

### See also

`me`, `spriteNum`

## currentTime

### Syntax

`sprite(whichSprite).currentTime`  
the `currentTime` of sprite *whichSprite*  
`sound(channelNum).currentTime`

### Description

Sprite and sound channel property; returns the current playing time, in milliseconds, for a sound sprite, QuickTime digital video sprite, or any Xtra that supports cue points. For a sound channel, returns the current playing time of the sound member currently playing in the given sound channel.

This property can be tested, but can only be set for traditional sound cast members (WAV, AIFF, SND). When this property is set, the range of allowable values is from zero to the `duration` of the member.

Shockwave Audio (SWA) sounds can appear as sprites in sprite channels, but they play sound in a sound channel. You should refer to SWA sound sprites by their sprite channel number rather than by a sound channel number.

### Example

This statement displays the current time, in seconds, of the sound sprite in sprite channel 10.

**Dot syntax:**

```
member("time").text = string(sprite (10).currentTime/ 1000)
```

**Verbose syntax:**

```
set the text of member "time" to (the currentTime of sprite 10) / 1000
```

This statement causes the sound playing in sound channel 2 to skip to the point 2.7 seconds from the beginning of the sound cast member:

```
sound(2).currentTime = 2700
```

**See also**

movieTime, duration

## currentTime (3D)

**Syntax**

```
member(whichCastmember).model(whichModel).keyframePlayer.\  
    currentTime  
member(whichCastmember).model(whichModel).bonesPlayer.\  
    currentTime
```

**Description**

3D keyframePlayer and bonesPlayer modifier property; indicates the local time of the motion being executed by the model. The currentTime property is measured in milliseconds, but it only corresponds to real time when the motion is playing at its original speed.

Playback of a motion by a model is the result of either a play() or queue() command. The scale parameter of the play() or queue() command is multiplied by the modifier's playRate property, and the resulting value is multiplied by the motion's original speed to determine how fast the model will execute the motion and how fast the motion's local time will run. So if the scale parameter has a value of 2 and the modifier's playRate property has a value of 3, the model will execute the motion six times as fast as its original speed and local time will run six times as fast as real time.

The currentTime property resets to the value of the cropStart parameter of the play() or queue() command at the beginning of each iteration of a looped motion.

**Example**

This statement shows the local time of the motion being executed by the model named Alien3.

```
put member("newalien").model("Alien3").keyframePlayer.currentTime  
-- 1393.8599
```

**See also**

play() (3D), queue() (3D), playlist

## currentTime (RealMedia)

**Syntax**

```
sprite(whichSprite).currentTime  
member(whichCastmember).currentTime  
sprite(whichSprite).currentTime = timeInMilliseconds  
member(whichCastmember).currentTime = timeInMilliseconds
```



### Description

RealMedia sprite or cast member property; allows you to get or set the current time of the RealMedia stream, in milliseconds. If the RealMedia cast member is not playing, the value of this property is 0, which is the default setting. This is a playback property, and it is not saved.

If the stream is playing when the `currentTime` property is set or changed, a seek action takes place, the stream rebuffers, and then playback resumes at the new time. If the stream is paused (`#paused` `mediaStatus` value) when `currentTime` is set or changed, the stream redraws the frame at the new time, and it resumes playback if `pausedAtStart` is set to `FALSE`. When the stream is paused or stopped in the RealMedia viewer, `mediaStatus` is `#paused`. When the stream is stopped by the Lingo `stop` command, `mediaStatus` is `#closed`. This property has no effect if the stream's `mediaStatus` value is `#closed`. When you set integer values, they are clipped to the range from 0 to the duration of the stream.

Setting `currentTime` is equivalent to invoking the `seek` command: `x.seek(n)` is the same as `x.currentTime = n`. Changing `currentTime` or calling `seek` will require the stream to be rebuffered.

### Examples

The following examples show that the current time of the sprite 2 and the cast member Real is 15,534 milliseconds (15.534 seconds) from the beginning of the stream.

```
put sprite(2).currentTime
-- 15534

put member("Real").currentTime
-- 15534
```

The following examples cause playback to jump 20,000 milliseconds (20 seconds) into the stream of sprite 2 and the cast member Real.

```
sprite(2).currentTime = 20000
member("Real").currentTime = 20000
```

### See also

`duration` (`RealMedia`), `seek`, `mediaStatus`

## cursor (command)

### Syntax

```
cursor [castNumber, maskCastNumber]
cursor whichCursor
cursor (member whichCursorCastMember)
```

### Description

Command; changes the cast member or built-in cursor that is used for a cursor and stays in effect until you turn it off by setting the cursor to 0.

- Use the syntax `cursor [castNumber, maskCastNumber]` to specify the number of a cast member to use as a cursor and its optional mask. The cursor's hot spot is the registration point of the cast member.

The cast member that you specify must be a 1-bit cast member. If the cast member is larger than 16 by 16 pixels, Director crops it to a 16-by-16-pixel square, starting in the upper left corner of the image. The cursor's hot spot is still the registration point of the cast member.

- Use the syntax `cursor whichCursor` to specify default system cursors. The term *whichCursor* must be one of the following integer values:

---

0*	No cursor set
-1	Arrow (pointer) cursor
1	I-beam cursor
2	Crosshair cursor
3*	Crossbar cursor
4	Watch cursor (Macintosh only)
200*	Blank cursor (hides cursor)

---

\* The Director player for Java does not support these cursor types and displays an arrow cursor instead.

- Use the syntax `cursor (member whichCursorCastMember)` for the custom cursors available through the Cursor Xtra.

Be sure not to confuse the syntax `cursor 1` with `cursor [1]`. The first selects the I-beam from the system cursor set; the second uses cast member 1 as the custom cursor.

**Note:** Although the Cursor Xtra allows cursors of different cast types, text cast members cannot be used as cursors.

During system events such as file loading, the operating system may display the watch cursor and then change to the pointer cursor when returning control to the application, overriding the `cursor` command settings from the previous movie. To use `cursor` at the beginning of any new movie that is loaded in a presentation using a custom cursor for multiple movies, store any special cursor resource number as a global variable that remains in memory between movies.

Cursor commands can be interrupted by an Xtra or other external agent. If the cursor is set to a value in Director and an Xtra or external agent takes control of the cursor, resetting the cursor to the original value has no effect because Director doesn't perceive that the cursor has changed. To work around this, explicitly set the cursor to a third value and then reset it to the original value.

### Example

This statement changes the cursor to a watch cursor on the Macintosh, and hourglass in Windows, whenever the value in the variable named `status` equals 1:

```
if status = 1 then cursor 4
```

This handler checks whether the cast member assigned to the variable is a 1-bit cast member and then uses it as the cursor if it is:

```
on myCursor someMember
  if the depth of member someMember = 1 then
    cursor[someMember]
  else
    beep
  end if
end
```

### See also

`cursor (sprite property), rollover()`

## cursor (sprite property)

### Syntax

```
sprite(whichSprite).cursor = [castNumber, maskCastNumber]  
set the cursor of sprite whichSprite to [castNumber, maskCastNumber]  
sprite(whichSprite).cursor = whichCursor  
set the cursor of sprite whichSprite to whichCursor
```

### Description

Sprite property; determines the cursor used when the pointer is over the sprite specified by the integer expression *whichSprite*. This property stays in effect until you turn it off by setting the cursor to 0. Use the `cursor` sprite property to change the cursor when the mouse pointer is over specific regions of the screen and to indicate regions where certain actions are possible when the user clicks on them.

When you set the `cursor` sprite property in a given frame, Director keeps track of the sprite rectangle to determine whether to alter the cursor. This rectangle persists when the movie enters another frame unless you set the `cursor` sprite property for that channel to 0.

- Use the syntax `cursor of sprite...[castNumber, maskCastNumber]` to specify the number of a cast member to use as a cursor and its optional mask.
- Use the syntax `cursor of sprite...whichCursor` to specify default system cursors. The term *whichCursor* must be one of the following integer values:

---

0*	No cursor set
-1	Arrow (pointer) cursor
1	I-beam cursor
2	Crosshair cursor
3*	Crossbar cursor
4	Watch cursor (Macintosh only)
200*	Blank cursor (hides cursor)

---

\* The Director player for Java does not support these cursor types and displays an arrow cursor instead.

To use custom cursors, set the `cursor` sprite property to a list containing the cast member to be used as a cursor or to the number that specifies a system cursor. In Windows, a cursor must be a cast member, not a resource; if a cursor is not available because it is a resource, Director displays the standard arrow cursor instead. For best results, don't use custom cursors when creating cross-platform movies.

If the sprite is a bitmap that has matte ink applied, the cursor changes only when the cursor is over the matte portion of the sprite.

When the cursor is over the location of a sprite that has been removed, rollover still occurs. Avoid this problem by not performing rollovers at these locations or by relocating the sprite up above the menu bar before deleting it.

On the Macintosh, you can use a numbered cursor resource in the current open movie file as the cursor by replacing *whichCursor* with the number of the cursor resource.

This property can be tested and set.

### Example

This statement changes the cursor that appears over sprite 20 to a watch cursor.

Dot syntax:

```
sprite(20).cursor = 4
```

Verbose syntax:

```
set the cursor of sprite 20 to 4
```

### See also

`cursor` (command)

## cursorSize

### Syntax

`member(whichCursorCastMember).cursorSize`  
the cursorSize of member *whichCursorCastMember*

### Description

Cursor cast member property; specifies the size of the animated color cursor cast member *whichCursorCastMember*.

---

Specify size:	For cursors up to:
16	16 by 16 pixels
32	32 by 32 pixels

---

Bitmap cast members smaller than the specified size are displayed at full size, and larger ones are scaled proportionally to the specified size.

The default value is 32 for Windows and 16 for the Macintosh. If you set an invalid value, an error message appears when the movie runs (but not when you compile).

This property can be tested and set.

### Example

This command resizes the animated color cursor stored in cast member 20 to 32 by 32 pixels.

Dot syntax:

```
member(20).cursorSize = 32
```

Verbose syntax:

```
set the cursorSize of member 20 = 32
```

## curve

### Syntax

*member.curve[curveListIndex]*

### Description

This property contains the `vertexList` of an individual curve (shape) from a vector shape cast member. You can use the `curve` property along with the `vertex` property to get individual vertices of a specific curve in a vector shape.

A `vertexList` is a list of vertices, and each vertex is a property list containing up to three properties: a `#vertex` property with the location of the vertex, a `#handle1` property with the location of the first control point for that vertex, and a `#handle2` property with the location of the second control point for that vertex. See `vertexList`.

### Examples

This statement displays the list of vertices of the third curve of vector shape member `SimpleCurves`:

```
put member("SimpleCurves").curve[3]
-- [[#vertex: point(113.0000, 40.0000), #handle1: point(32.0000, 10.0000),
    #handle2: point(-32.0000, -10.0000)], [#vertex: point(164.0000, 56.0000)]]
```

This statement moves the first vertex of the first curve in a vector shape down and to the right by 10 pixels:

```
member(1).curve[1].vertex[1] = member(1).curve[1].vertex[1] + point(10, 10)
```

The following code moves a sprite to the location of the first vertex of the first curve in a vector shape. The vector shape's `originMode` must be set to `#topLeft` for this to work.

```
vertexLoc = member(1).curve[1].vertex[1]
spriteLoc = mapMemberToStage(sprite(3), vertexLoc)
sprite(7).loc = spriteLoc
vertex, vertexList
```

## date() (system clock)

### Syntax

the abbr date  
the abbrev date  
the abbreviated date  
the date  
the long date  
the short date

### Description

Function; returns the current date in the system clock in one of three formats: abbreviated, long, or short (default). The abbreviated format can also be referred to as `abbrev` and `abbr`.

In Java, the `date` function is available, but it doesn't accept `abbrev`, `long`, or `short` modifiers. When the movie plays back as an applet, the date's format is `MM/DD/YY`, where `MM` represents the month, `DD` represents the day, and `YY` represents the last two digits of the current year. For the months January through September, the value for `MM` is a single digit.

The format Director uses for the date varies, depending on how the date is formatted on the computer.

- In Windows, you can customize the date display by using the International control panel. (Windows stores the current short date format in the System.ini file. Use this value to determine what the parts of the short date indicate.)
- On the Macintosh, you can customize the date display by using the Date and Time control panel.

### Examples

This statement displays the abbreviated date:

```
put the abbreviated date
-- "Sat, Sep 7, 1991"
```

This statement displays the long date:

```
put the long date
-- "Saturday, September 7, 1991"
```

This statement displays the short date:

```
put the short date
-- "9/7/91"
```

This statement tests whether the current date is January 1 by checking whether the first four characters of the date are 1/1. If it is January 1, the alert “Happy New Year!” appears:

```
if char 1 to 4 of the date = "1/1/" then alert "Happy New Year!"
```

**Note:** The three date formats vary, depending on the country for which your operating system was designed. These examples are for the United States. Use the `date` object to create and manipulate dates in a standard format.

### See also

`time()`, `date()` (formats), `systemDate`

## date() (formats)

### Syntax

```
date(ISOFormatString)
date(ISOFormatInteger)
date(ISOFormatIntegerYear, ISOFormatIntegerMonth, ISOFormatIntegerDay)
```

### Description

Function and data type; creates a standard, formatted date object instance for use with other date object instances in arithmetic operations and for use in manipulating dates across platforms and in international formats.

When creating the date, use four digits for the year, two digits for the month, and two digits for the day. The following expressions are equivalent:

integer:	set vacationStart = date(19980618)
string:	set vacationStart = date("19980618")
comma separated:	set vacationStart = date(1998, 06, 18)

Addition and subtraction operations on the date are interpreted as the addition and subtraction of days.

The individual properties of the date object instance returned are:

---

<code>#year</code>	Integer representing the year
<code>#month</code>	Integer representing the month of the year
<code>#day</code>	Integer representing the day of the month

---

### Examples

These statements create and determine the number of days between two dates:

```
myBirthDay = date(19650712)
yourBirthDay = date(19450529)
put "There are" && abs(yourBirthDay - myBirthDay) && "days between our
  birthdays."
```

These statements access an individual property of a date:

```
myBirthDay = date(19650712)
put "I was born in month number"&&myBirthDay.month
```

### See also

`date()` (system clock)

## deactivateApplication

### Syntax

```
on deactivateApplication
```

### Description

Built-in handler; runs when the projector is sent to the background. This handler is useful when a projector runs in a window and the user can send it to the background to work with other applications. Any MIAWs running in the projector can also make use of this handler.

During authoring, this handler is called only if Animate in Background is turned on in General Preferences.

On Windows, this handler is not called if the projector is merely minimized and no other application is brought to the foreground.

### Example

This handler plays a sound each time the user sends the projector to the background:

```
on deactivateApplication
  sound(1).queue(member("closeSound"))
  sound(1).play()
end
```

### See also

`add (3D texture)`, `activeCastLib`, `on deactivateWindow`

## on deactivateWindow

### Syntax

```
on deactivateWindow
    statement(s)
end
```

### Description

System message and event handler; contains statements that run when the window that the movie is playing in is deactivated. The `on deactivate` event handler is a good place for Lingo that you want executed whenever a window is deactivated.

### Example

This handler plays the sound `Snore` when the window that the movie is playing in is deactivated:

```
on deactivateWindow
    puppetSound 2, "Snore"
end
```

## debug

### Syntax

```
member(whichCastmember).model(whichModel).debug
```

### Description

3D model property; indicates whether the bounding sphere and local axes of the model are displayed.

### Example

This statement sets the `debug` property of the model `Dog` to `TRUE`.

```
member("ParkScene").model("Dog").debug = TRUE
```

### See also

`boundingSphere`

## debugPlaybackEnabled

### Syntax

```
the debugPlaybackEnabled
```

### Description

Property; in Windows, opens a Message window for debugging purposes in Shockwave and projectors. It does not have any effect when used in the Director application. Once the Message window is closed, it cannot be reopened for a particular Shockwave or projector session. If more than one Shockwave movie uses this Lingo in a single browser, only the first will open a Message window, and the Message window will be tied to the first movie alone.

On the Macintosh, rather than a Message window being opened, a log file is generated to allow Lingo `put` statements to output data for debugging purposes. This file is located in the Shockwave folder at `HardDrive/System Folder/Extensions/Macromedia/Shockwave`.

To open this Message window, set the `debugPlaybackEnabled` property to `TRUE`. To close the window, set the `debugPlaybackEnabled` property to `FALSE`.



### Example

This statement opens the Message window in either Shockwave or a projector:

```
the debugPlaybackEnabled = TRUE
```

## decayMode

### Syntax

```
member(whichCastmember).camera(whichCamera).fog.decayMode  
sprite(whichSprite).camera{( index )}.fog.decayMode
```

### Description

3D property; indicates the manner in which fog density builds from minimum to maximum density when the camera's `fog.enabled` property is set to `TRUE`.

The following are the possible values for this property:

- `#linear`: the fog density is linearly interpolated between `fog.near` and `fog.far`.
- `#exponential`: `fog.far` is the saturation point; `fog.near` is ignored.
- `#exponential2`: `fog.near` is the saturation point; `fog.far` is ignored.

The default setting for this property is `#exponential`.

### Example

This statement sets the `decayMode` property of the fog of the camera `Defaultview` to `#linear`. If the fog's `enabled` property is set to `TRUE`, the density of the fog will steadily increase between the distances set by the fog's `near` and `far` properties. If the `near` property is set to 100 and the `far` property is set to 1000, the fog will begin 100 world units in front of the camera and gradually increase in density to a distance of 1000 world units in front of the camera.

```
member("3d world").camera("Defaultview").fog.decayMode = #linear
```

### See also

`fog`, `near (fog)`, `far (fog)`, `enabled (fog)`

## defaultRect

### Syntax

```
member(whichFlashOrVectorShapeMember).defaultRect  
the defaultRect of member whichFlashOrVectorShapeMember
```

### Description

Cast member property; controls the default size used for all new sprites created from a Flash movie or vector shape cast member. The `defaultRect` setting also applies to all existing sprites that have not been stretched on the Stage. You specify the property values as a Director rectangle; for example, `rect(0,0,32,32)`.

The `defaultRect` member property is affected by the cast member's `defaultRectMode` member property. The `defaultRectMode` property is always set to `#Flash` when a movie is inserted into a cast, which means the original `defaultRect` setting is always the size of the movie as it was originally created in Flash. Setting `defaultRect` after that implicitly changes the cast member's `defaultRectMode` property to `#fixed`.

This property can be tested and set.

### Example

This handler accepts a cast reference and a rectangle as parameters. It then searches the specified cast for Flash cast members and sets their `defaultRect` property to the specified rectangle.

```
on setDefaultFlashRect whichCast, whichRect
  repeat with i = 1 to the number of members of castLib whichCast
    if member(i, whichCast).type = #flash then
      member(i, whichCast).defaultRect = whichRect
    end if
  end repeat
end
```

### See also

`defaultRectMode`, `flashRect`

## defaultRectMode

### Syntax

`member(whichVectorOrFlashMember).defaultRectMode`  
the `defaultRectMode` of member *whichVectorOrFlashMember*

### Description

Cast member property; controls how the default size is set for all new sprites created from Flash movie or vector shape cast members. You specify the property value as a Director rectangle; for example, `rect(0,0,32,32)`.

The `defaultRectMode` property does not set the actual size of a Flash movie's default rectangle; it only determines how the default rectangle is set. The `defaultRectMode` member property can have these values:

- `#flash` (default)—Sets the default rectangle using the size of the movie as it was originally created in Flash.
- `#fixed`—Sets the default rectangle using the fixed size specified by the `defaultRect` member property.

The `defaultRect` member property is affected by the cast member's `defaultRectMode` member property. The `defaultRectMode` property is always set to `#flash` when a movie is inserted into a cast, which means the original `defaultRect` setting is always the size of the movie as it was originally created in Flash. Setting `defaultRect` after that implicitly changes the cast member's `defaultRectMode` property to `#fixed`.

This property can be tested and set.

### Example

This handler accepts a cast reference and a rectangle as parameters. It then searches the specified cast for Flash cast members, sets their `defaultRectMode` property to `#fixed`, and then sets their `defaultRect` property to `rect(0,0,320,240)`.

```
on setDefaultRectSize whichCast
  repeat with i = 1 to the number of members of castLib whichCast
    if member(i, whichCast).type = #flash then
      member(i, whichCast).defaultRectMode = #fixed
      member(i, whichCast).defaultRect = rect(0,0,320,240)
    end if
  end repeat
end
```

**See also**

flashRect, defaultRect

## delay

**Syntax**

`delay numberOfTicks`

**Description**

Command; pauses the playhead for a given amount of time. The integer expression *numberOfTicks* specifies the number of ticks to wait, where each tick is 1/60 of a second. The only mouse and keyboard activity possible during this time is stopping the movie by pressing Control+Alt+period (Windows) or Command+period (Macintosh). Because it increases the time of individual frames, the `delay` command is useful for controlling the playback rate of a sequence of frames.

The `delay` command can be applied only when the playhead is moving. However, when `delay` is in effect, handlers still run: only the playhead halts, not script execution. Place scripts that use the `delay` command in either an `on enterFrame` or `on exitFrame` handler.

To mimic the behavior of a halt in a handler when the playhead is not moving, use the `startTimer` command or assign the current value of `timer` to a variable and wait for the specified amount of time to pass before exiting the frame.

**Examples**

This handler delays the movie for 2 seconds when the playhead exits the current frame:

```
on exitFrame
    delay 2 * 60
end
```

This handler, which can be placed in a frame script, delays the movie a random number of ticks:

```
on keyDown
    if the key = RETURN then delay random(180)
end
```

**Example**

The first of these handlers sets a timer when the playhead leaves a frame. The second handler, assigned to the next frame, loops in the frame until the specified amount of time passes:

```
--script for first frame
on exitFrame
    global gTimer
    set gTimer = the ticks
end

--script for second frame
on exitFrame
    global gTimer
    if the ticks < gTimer + (10 * 60) then
        go to the frame
    end if
end
```

**See also**

startTimer, ticks, timer

## delete

### Syntax

```
delete chunkExpression
```

### Description

Command; deletes the specified chunk expression (character, word, item, or line) in any string container. Sources of strings include field cast members and variables that hold strings.

To see an example of `delete` used in a completed movie, see the Text movie in the Learning/Lingo Examples folder inside the Director application folder.

### Examples

This statement deletes the first word of line 3 in the field cast member Address:

```
delete word 1 of line 3 of member "Address"
```

The same chunk of text may also be deleted with the syntax:

```
delete member("Address").line[3].word[1]
```

This statement deletes the first character of the string in the variable `bidAmount` if that character is the dollar sign (“\$”):

```
if bidAmount.char[1] = "$" then delete bidAmount.char[1]
```

### See also

`char...of`, `field`, `item...of`, `line...of`, `word...of`, `hilite` (cast member property), `paragraph`

## deleteAll

### Syntax

```
list.deleteAll()  
deleteAll list
```

### Description

List command; deletes all items in the specified list without changing the list type.

### Example

This statement deletes every item in the list named `propList`:

```
propList.deleteAll()
```

## deleteAt

### Syntax

```
list.deleteAt(number)  
deleteAt list, number
```

### Description

List command; deletes the item in the position specified by *number* from the linear or property list specified by *list*.

The `deleteAt` command checks whether an item is in a list; if you try to delete an object that isn't in the list, Director displays an alert.

### Example

This statement deletes the second item from the list named designers, which contains [gee, kayne, ohashi]:

```
designers = ["gee", "kayne", "ohashi"]  
designers.deleteAt(2)
```

The result is the list [gee, ohashi].

This handler checks whether an object is in a list before attempting to delete it:

```
on myDeleteAt theList, theIndex  
  if theList.count < theIndex then  
    beep  
  else  
    theList.deleteAt(theIndex)  
  end if  
end
```

### See also

addAt

## deleteCamera

### Syntax

```
member(whichCastmember).deleteCamera(cameraName)  
member(whichCastmember).deleteCamera(index)  
sprite(whichSprite).deleteCamera(cameraOrIndex)
```

### Description

3D command; in a cast member, this command removes the camera from the cast member and the 3D world. Children of the camera are removed from the 3D world but not deleted.

It is not possible to delete the default camera of the cast member.

In a sprite, this command removes the camera from the sprite's list of cameras. The camera is not deleted from the cast member.

### Examples

This statement deletes two cameras from the cast member named Room: first the camera named Camera06, and then camera 1.

```
member("Room").deleteCamera("Camera06")  
member("Room").deleteCamera(1)
```

This statement removes two cameras from the list of cameras for sprite 5: first the second camera in the list, then the camera named Camera06

```
sprite(5).deleteCamera(2)  
sprite(5).deleteCamera(member("Room").camera("Camera06"))
```

### See also

newCamera, addCamera, cameraCount()

## deleteFrame

### Syntax

`deleteFrame`

### Description

Command; deletes the current frame and makes the next frame the new current frame during a Score generation session only.

### Example

The following handler checks whether the sprite in channel 10 of the current frame has gone past the right edge of a 640-by-480-pixel Stage and deletes the frame if it has:

```
on testSprite
  beginRecording
    if sprite(10).locH > 640 then deleteFrame
  endRecording
end
```

### See also

`beginRecording`, `endRecording`, `updateFrame`

## deleteGroup

### Syntax

```
member(whichCastmember).deleteGroup(whichGroup)
member(whichCastmember).deleteGroup(index)
```

### Description

3D command; removes the group from the cast member and the 3D world. Children of the group are removed from the 3D world but not deleted.

It is not possible to delete the group named World, which is the default group.

### Example

The first line of this example deletes the group Dummy16 from the cast member Scene. The second line deletes the third group of Scene.

```
member("Scene").deleteGroup("Dummy16")
member("Scene").deleteGroup(3)
```

### See also

`newGroup`, `child`, `parent`

## deleteLight

### Syntax

```
member(whichCastmember).deleteLight(whichLight)  
member(whichCastmember).deleteLight(index)
```

### Description

3D command; removes the light from the cast member and the 3D world. Children of the light are removed from the 3D world but not deleted.

### Examples

These examples delete lights from the cast member named Room.

```
member("Room").deleteLight("ambientRoomLight")  
member("Room").deleteLight(6)
```

### See also

newLight

## deleteModel

### Syntax

```
member(whichCastmember).deleteModel(whichModel)  
member(whichCastmember).deleteModel(index)
```

### Description

3D command; removes the model from the cast member and the 3D world. Children of the model are removed from the 3D world but not deleted.

### Examples

The first line of this example deletes the model named Player3 from the cast member named gbWorld. The second line deletes the ninth model of gbWorld.

```
member("gbWorld").deleteModel("Player3")  
member("gbWorld").deleteModel(9)
```

### See also

newModel

## deleteModelResource

### Syntax

```
member(whichCastmember).deleteModelResource(whichModelResource)  
member(whichCastmember).deleteModelResource(index)
```

### Description

3D command; removes the model resource from the cast member and the 3D world.

Models using the deleted model resource become invisible, because they lose their geometry, but they are not deleted or removed from the world.

### Example

These examples delete two model resources from the cast member named StreetScene.

```
member("StreetScene").deleteModelResource("HouseB")  
member("StreetScene").deleteModelResource(3)
```

### See also

newModelResource, newMesh

## deleteMotion

### Syntax

```
member(whichCastmember).deleteMotion(whichMotion)  
member(whichCastmember).deleteMotion(index)
```

### Description

3D command; removes the motion from the cast member.

### Examples

The first line of this example deletes the motion named BackFlip from the cast member named PicnicScene. The second line deletes the fifth motion in PicnicScene.

```
member("PicnicScene").deleteMotion("BackFlip")  
member("PicnicScene").deleteMotion(5)
```

### See also

`newMotion()`, `removeLast()`

## deleteOne

### Syntax

```
list.deleteOne(value)  
deleteOne list, value
```

### Description

List command; deletes a value from a linear or property list. For a property list, `deleteOne` also deletes the property associated with the deleted value. If the value appears in the list more than once, `deleteOne` deletes only the first occurrence.

Attempting to delete a property has no effect.

### Example

The first statement creates a list consisting of the days Tuesday, Wednesday, and Friday. The second statement deletes the name Wednesday from the list.

```
days = ["Tuesday", "Wednesday", "Friday"]  
days.deleteOne("Wednesday")  
put days
```

The `put days` statement causes the Message window to display the result:

```
-- ["Tuesday", "Friday"].
```

## deleteProp

### Syntax

```
list.deleteProp(item)  
deleteProp list, item
```

### Description

List command; deletes the specified item from the specified list.

- For linear lists, replace *item* with the number identifying the list position of the item to be deleted. The `deleteProp` command for linear lists is the same as the `deleteAt` command. If the number is greater than the number of items in the list, a script error occurs.



- For property lists, replace *item* with the name of the property to be deleted. Deleting a property also deletes its associated value. If the list has more than one of the same property, only the first property in the list is deleted.

#### Example

This statement deletes the color property from the list [#height:100, #width: 200, #color: 34, #ink: 15], which is called `spriteAttributes`:

```
spriteAttributes.deleteProp(#color)
```

The result is the list [#height:100, #width: 200, #ink: 15].

#### See also

`deleteAt`

## deleteShader

#### Syntax

```
member(whichCastmember).deleteShader(whichShader)
member(whichCastmember).deleteShader(index)
```

#### Description

3D command; removes the shader from the cast member.

#### Example

The first line of this example deletes the shader Road from the cast member named StreetScene. The second line deletes the third shader of StreetScene.

```
member("StreetScene").deleteShader("Road")
member("StreetScene").deleteShader(3)
```

#### See also

`newShader`, `shaderList`

## deleteTexture

#### Syntax

```
member(whichCastmember).deleteTexture(whichTexture)
member(whichCastmember).deleteTexture(index)
```

#### Description

3D command; removes the shader from the cast member.

#### Example

The first line of this example deletes the texture named Sky from the cast member named PicnicScene. The second line deletes the fifth texture of PicnicScene.

```
member("PicnicScene").deleteTexture("Sky")
member("PicnicScene").deleteTexture(5)
```

#### See also

`newTexture`

## deleteVertex()

### Syntax

```
member(memberRef).deleteVertex(indexToRemove)  
deleteVertex(member memberRef, indexToRemove)
```

### Description

Vector shape command; removes an existing vertex of a vector shape cast member in the index position specified.

### Example

This line removes the second vertex point in the vector shape Archie:

```
member("Archie").deleteVertex(2)
```

### See also

addVertex, moveVertex(), originMode, vertexList

## density

### Syntax

```
member(whichCastmember).shader(whichShader).density  
member(whichCastmember).model(whichModel).shader.density  
member(whichCastmember).model(whichModel).shaderList[[index]].\  
density
```

### Description

3D #engraver and #newsprint shader property; adjusts the number of lines or dots used to create the effects of these specialized shader types. Higher values result in more lines or dots.

For #engraver shaders, this property adjusts the number of lines used to create the image. The range is 0 to 100 and the default value is 40.

For #newsprint shaders, this property adjusts the number of dots used to create the image. The value can be from 0 to 100 and the default value is 45.

### Example

The following statement sets the density property of the shader named EngShader to 10. The lines used by this #engraver shader to create its stylized image will be coarse and far apart.

```
member("scene").shader("EngShader").density = 10
```

The following statement sets the density property of the shader gbShader to 100. The dots used by this #newsprint shader to create its stylized image will be very fine and close together.

```
member("scene").shader("gbShader").density = 100
```

### See also

newShader

## depth

### Syntax

*imageObject*.depth  
member(*whichCastMember*).depth  
the depth of member *whichCastMember*

### Description

Image object or bitmap cast member property; displays the color depth of the given image object or bitmap cast member.

Depth	Number of Colors
1	Black and white
2	4 colors
4, 8	16 or 256 palette-based colors, or gray levels
16	Thousands of colors
32	Millions of colors

This property can be tested but not set.

### Examples

This statement displays the color depth of the image object stored in the variable `newImage`. The output appears in the Message window.

```
put newImage.depth
```

This statement displays the color depth of the cast member `Shrine` in the Message window:

```
put member("Shrine").depth
```

## depth (3D)

### Syntax

member(*whichCastmember*).model(*whichModel*).sds.depth

### Description

3D subdivision surfaces (`sds`) modifier property; specifies the maximum number of levels of resolution that the model can display when using the `sds` modifier.

If the `sds` modifier's `error` and `tension` settings are low, increasing the `depth` property will have a more pronounced effect on the model's geometry.

The `sds` modifier cannot be used with the `inker` or `toon` modifiers, and you should be careful when using the `sds` modifier with the `lod` modifier.

### Example

This statement sets the `depth` property of the `sds` modifier for the model named `Baby` to 3. If the `sds` modifier's `error` and `tension` settings are low, this will cause a very pronounced effect on `Baby`'s geometry.

```
member("Scene").model("Baby").sds.depth = 3
```

### See also

`sds` (modifier), `error`, `tension`

## depthBufferDepth

### Syntax

```
getRendererServices().depthBufferDepth
```

### Description

3D `rendererServices` property; indicates the precision of the hardware depth buffer of the user's system. The value is either 16 or 24, depending on the user's hardware settings.

### Example

This statement shows that the `depthBufferDepth` value of the user's video card is 16:

```
put getRendererServices().depthBufferDepth
-- 16
```

### See also

```
getRendererServices(), getHardwareInfo(), colorBufferDepth
```

## deskTopRectList

### Syntax

```
the deskTopRectList
```

### Description

System property; displays the size and position on the desktop of the monitors connected to a computer. This property is useful for checking whether objects such as windows, sprites, and pop-up windows appear entirely on one screen.

The result is a list of rectangles, where each rectangle is the boundary of a monitor. The coordinates for each monitor are relative to the upper left corner of monitor 1, which has the value (0,0). The first set of rectangle coordinates is the size of the first monitor. If a second monitor is present, a second set of coordinates shows where the corners of the second monitor are relative to the first monitor.

This property can be tested but not set.

### Examples

This statement tests the size of the monitors connected to the computer and displays the result in the Message window:

```
put the deskTopRectList
-- [rect(0,0,1024,768), rect(1025, 0, 1665, 480)]
```

The result shows that the first monitor is 1024 by 768 pixels and the second monitor is 640 by 480 pixels.

This handler tells how many monitors are in the current system:

```
on countMonitors
    return the deskTopRectList.count
end
```

# diffuse

## Syntax

```
member(whichCastmember).shader(whichShader).diffuse  
member(whichCastmember).model(whichModel).shader.diffuse  
member(whichCastmember).model(whichModel).shaderList[[index]].\  
diffuse
```

## Description

3D #standard shader property; indicates a color that is blended with the first texture of the shader when the following conditions are met:

- the shader's `useDiffuseWithTexture` property is set to `TRUE`, and either
- the `blendFunction` property of the shader is set to `#add` or `#multiply`, or
- the `blendFunction` property of the shader is set to `#blend`, the `blendSource` property of the shader is set to `#constant`, and the value of the `blendConstant` property of the shader is less than 100.

The default value is of this property is `rgb( 255, 255, 255 )`.

## Example

This statement sets the `diffuse` property of the shader named `Globe` to `rgb(255, 0, 0)`.

```
member("MysteryWorld").shader("Globe").diffuse = rgb(255, 0, 0)
```

## See also

`diffuseColor`, `useDiffuseWithTexture`, `blendFunction`, `blendSource`, `blendConstant`

# diffuseColor

## Syntax

```
member(whichCastmember).diffuseColor
```

## Description

3D cast member property; indicates a color that is blended with the first texture of the first shader of the cast member when the following conditions are met:

- the shader's `useDiffuseWithTexture` property is set to `TRUE`, and either
- the `blendFunction` property of the shader is set to `#add` or `#multiply`, or
- the `blendFunction` property of the shader is set to `#blend`, the `blendSource` property of the shader is set to `#constant`, and the value of the `blendConstant` property of the shader is less than 100.

The default value is of the `diffuseColor` property is `rgb( 255, 255, 255 )`.

## Example

This statement sets the `diffuseColor` property of the cast member named `Room` to `rgb(255, 0, 0)`.

```
member("Room").diffuseColor = rgb(255, 0, 0)
```

## See also

`diffuse`, `useDiffuseWithTexture`, `blendFunction`, `blendSource`, `blendConstant`

## diffuseLightMap

### Syntax

```
member(whichCastmember).shader(whichShader).diffuseLightMap  
member(whichCastmember).model(whichModel).shader.diffuseLightMap  
member(whichCastmember).model(whichModel).shaderList[[index]].\  
diffuseLightMap
```

### Description

3D #standard shader property; specifies the texture to use for diffuse light mapping.

When you set this property, the following properties are automatically set:

- The second texture layer of the shader is set to the texture you specified.
- The value of `textureModelList[2]` is set to #diffuse.
- The value of `blendFunctionList[2]` is set to #multiply.
- The value of `blendFunctionList[1]` is set to #replace.

### Example

This statement sets the texture named Oval as the `diffuseLightMap` property of the shader used by the model named GlassBox.

```
member("3DPlanet").model("GlassBox").shader.diffuseLightMap = \  
member("3DPlanet").texture("Oval")
```

### See also

`blendFunctionList`, `textureModelList`, `glossMap`, `region`, `specularLightMap`

## digitalVideoTimeScale

### Syntax

```
the digitalVideoTimeScale
```

### Description

System property; determines the time scale, in units per second, that the system uses to track digital video cast members.

The `digitalVideoTimeScale` property can be set to any value you choose.

The value of the property determines the fraction of a second that is used to track the video, as in the following examples:

- 100—The time scale is 1/100 of a second (and the movie is tracked in 100 units per second).
- 500—The time scale is 1/500 of a second (and the movie is tracked in 500 units per second).
- 0—Director uses the time scale of the movie that is currently playing.

Set `digitalVideoTimeScale` to precisely access tracks by ensuring that the system's time unit for video is a multiple of the digital video's time unit. Set the `digitalVideoTimeScale` property to a higher value to enable finer control of video playback.

This property can be tested and set.

### Example

This statement sets the time scale that the system uses to measure digital video to 600 units per second:

```
the digitalVideoTimeScale to 600
```

# digitalVideoType

## Syntax

member(*whichCastMember*).digitalVideoType  
the digitalVideoType of member *whichCastMember*

## Description

Cast member property; indicates the format of the specified digital video. Possible values are #quickTime or #videoForWindows.

This property can be tested but not set.

## Example

The following statement tests whether the cast member Today's Events is a QuickTime or AVI (Audio-Video Interleaved) digital video and displays the result in the Message window:

```
put member("Today's Events").digitalVideoType
```

## See also

quickTimeVersion()

# direction

## Syntax

member(*whichCastmember*).modelResource(*whichModelResource*).\  
emitter.direction

## Description

3D emitter property; a vector that indicates the direction in which the particles of a particle system are emitted. A particle system is a model resource whose type is #particle.

The primary direction of particle emission is the vector set by the emitter's direction property. However, the direction of emission of a given particle will deviate from that vector by a random angle between 0 and the value of the emitter's angle property.

Setting direction to vector(0,0,0) causes the particles to be emitted in all directions.

The default value of this property is vector(1,0,0).

## Example

In this example, ThermoSystem is a model resource whose type is #particle. This statement sets the direction property of ThermoSystem's emitter to vector(1, 0, 0), which causes the particles of ThermoSystem to be emitted into a conical region whose axis is the X axis of the 3D world.

```
member("Fires").modelResource("ThermoSystem").emitter.\  
direction = vector(1,0,0)
```

## See also

emitter, angle

## directionalColor

### Syntax

`member(whichCastmember).directionalColor`

### Description

3D cast member property; indicates the RGB color of the default directional light of the cast member.

The default value of this property is `rgb(255, 255, 255)`.

### Example

This statement sets the `directionalColor` property of the cast member named Room to `rgb(0, 255, 0)`. The default directional light of the cast member will be green. This property can also be set in the Property inspector.

```
member("Room").directionalcolor = rgb(0, 255, 0)
```

### See also

`directionalPreset`

## directionalPreset

### Syntax

`member(whichCastmember).directionalPreset`

### Description

3D cast member property; indicates the direction from which the default directional light shines, relative to the camera of the sprite.

Changing the value of this property results in changes to the `position` and `rotation` properties of the light's transform.

Possible values of `directionalPreset` include the following:

- `#topLeft`
- `#topCenter`
- `#topRight`
- `#middleLeft`
- `#middleCenter`
- `#middleRight`
- `#bottomLeft`
- `#bottomCenter`
- `#bottomRight`
- `#None`

The default value of this property is `#topCenter`.

### Example

This statement sets the `directionalPreset` property of the cast member named Room to `#middleCenter`. This points the default light of Room so it will shine on the middle center the current view of the camera of the sprite. This property can also be set in the Property inspector.

```
member("Room").directionalpreset = #middleCenter
```

### See also

`directionalColor`



## directToStage

### Syntax

`member(whichCastMember).directToStage`  
the `directToStage` of member *whichCastMember*  
`sprite(whichSprite).directToStage`  
the `directToStage` of sprite *whichSprite*

### Description

Sprite property and member property; determines the layer where a digital video, animated GIF, vector shape, 3D, or Flash Asset cast member plays. If this property is `TRUE` (1), the cast member plays in front of all other layers on the Stage, and ink effects have no affect. If this property is `FALSE` (0), the cast member can appear in any layer of the Stage's animation planes, and ink effects affect the appearance of the sprite.

- Use the syntax `member(whichCastMember).directToStage` for digital video or animated GIFs.
- Use the syntax `sprite(whichSprite).directToStage` for Flash or vector shapes.
- Use either syntax for 3D cast members or sprites.

Using this property improves the playback performance of the cast member or sprite.

No other cast member can appear in front of a `directToStage` sprite. Also, ink effects do not affect the appearance of a `directToStage` sprite.

When a sprite's `directToStage` property is `TRUE`, Director draws the sprite directly to the screen without first compositing it in the Director offscreen buffer. The result can be similar to the trails ink effect of the Stage.

Explicitly refresh a trailed area by turning the `directToStage` property off and on, using a full-screen transition, or “wiping” another sprite across this area. (In Windows, if you don't do this, you can branch to another similar screen, and the video may not completely disappear.)

To see an example of `directToStage` used in a completed movie, see the QT and Flash movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This statement makes the QuickTime movie The Residents always play in the top layer of the Stage:

```
member("The Residents").directToStage = 1
```

## disableImagingTransformation

### Syntax

the `disableImagingTransformation`

### Description

Imaging Lingo property; When `TRUE`, this property prevents Director from automatically taking Stage scrolling or zooming into account when `(the stage).image` is used. When the `disableImagingTransformation` property is `FALSE`, Director will always capture the image of the stage as if the stage window was zoomed at 100% and was not scrolled out from the center of the stage window. Then this property is `TRUE`, zooming and scrolling of the Stage will affect the appearance of the image captured by using `(the stage).image`.

## displayFace

### Syntax

```
member(whichTextCastmember).displayFace  
member(which3DCastmember).modelResource(whichModelResource).\  
    displayFace
```

### Description

3D text property; a linear list indicating which face or faces of the 3D text to display. Possible values include `#front`, `#tunnel`, and `#back`. You can show any combination of faces, and the list can be in any order.

The default value of this property is `[#front, #back, #tunnel]`.

For text cast members, this is a member property. For extruded text in a 3D cast member, this is a model resource property.

### Example

In this example, the cast member named `Rugsign` is a text cast member. This statement sets the `displayFace` property of `Rugsign` to `[#tunnel]`. When `Rugsign` is displayed in 3D mode, its front and back faces will not appear.

```
member("Rugsign").displayFace = [#tunnel]
```

In this example, the model resource of the model named `Slogan` is extruded text. This statement sets the `displayFace` property of `Slogan`'s model resource to `[#back, #tunnel]`. The front face of `Slogan` will not be drawn.

```
member("scene").model("Slogan").resource.displayFace = \  
    [#back, #tunnel]
```

### See also

`extrude3D`, `displayMode`

## displayMode

### Syntax

```
member(whichTextCastmember).displayMode
```

### Description

Text cast member property; specifies whether the text will be rendered as 2D text or 3D text.

If this property is set to `#Mode3D`, the text is shown in 3D. You can set the 3D properties (such as `displayFace` and `bevelDepth`) of the text, as well as the usual text properties (such as `text` and `font`). The sprite containing this cast member becomes a 3D sprite.

If this property is set to `#ModeNormal`, the text is shown in 2D.

The default value of this property is `#ModeNormal`.

### Example

In this example, the cast member named `Logo` is a text cast member. This statement causes `Logo` to be displayed in 3D.

```
member("Logo").displayMode = #mode3D
```

### See also

`extrude3D`

# displayRealLogo

## Syntax

```
sprite(whichSprite).displayRealLogo  
member(whichCastmember).displayRealLogo
```

## Description

RealMedia sprite or cast member property; allows you to set or get whether the RealNetworks logo is displayed (TRUE) or not (FALSE). When set to TRUE, this property displays the RealNetworks logo in the RealMedia viewer at the beginning of the stream, when the video is stopped, or when the video is rewound.

The default value of this property is TRUE (1). Integer values other than 1 or 0 are treated as TRUE.

## Examples

The following examples show that the `displayRealLogo` property for sprite 2 and the cast member Real is set to TRUE, which means that the RealNetworks logo is displayed when the movie starts to play and when it is stopped or rewound.

```
put sprite(2).displayRealLogo  
-- 1  
  
put member("Real").displayRealLogo  
-- 1
```

The following examples set the `displayRealLogo` property for sprite 2 and the cast member Real to FALSE, which means that the RealNetworks logo is not displayed.

```
sprite(2).displayRealLogo = FALSE  
member("Real").displayRealLogo = FALSE
```

# distanceTo()

## Syntax

```
vector1.distanceTo(vector2)
```

## Description

3D vector method; returns the distance in world units between two vectors.

## Example

There are three vectors in this example. The distance from Vec1 to Vec2 is 100.0000 world units. The distance from Vec1 to Vec3 is 141.4214 world units.

```
Vec1 = vector(100, 0, 0)  
Vec2 = vector(100, 100, 0)  
Vec3 = vector(100, 100, 100)  
put Vec1.distanceTo(Vec2)  
-- 100.0000  
put Vec1.distanceTo(Vec3)  
-- 141.4214
```

## See also

magnitude

## distribution

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
    emitter.distribution
```

### Description

3D emitter property; indicates how the particles of a particle system are distributed across the emitter's region at their creation. The possible values of this property are `#gaussian` or `#linear`. The default value is `#linear`.

### Example

In this example, `ThermoSystem` is a model resource whose type is `#particle`. This statement sets the `distribution` property of `ThermoSystem`'s emitter to `#linear`, which causes the particles of `ThermoSystem` to be evenly distributed across their origin region at their birth.

```
member("Fires").modelResource("ThermoSystem").emitter.\
    distribution = #linear
```

### See also

`emitter`, `region`

## dither

### Syntax

```
member(whichMember).dither
the dither of member whichMember
```

### Description

Bitmap cast member property; dithers the cast member when it is displayed at a color depth of 8 bits or less (256 colors) if the display must show a color gradation not in the cast member (`TRUE`), or tells Director to choose the nearest color out of those available in the current palette (`FALSE`).

For both performance and quality reasons, you should set `dither` to `TRUE` only when higher display quality is necessary. Dithering is slower than remapping, and artifacts may be more apparent when animating over a dithered image.

If the color depth is greater than 8 bits, this property has no effect.

### See also

`depth`

## do

### Syntax

```
do stringExpression
```

### Description

Command; evaluates *stringExpression* and executes the result as a Lingo statement. This command is useful for evaluating expressions that the user has typed and for executing commands stored in string variables, fields, arrays, and files.

Using uninitialized local variables within a `do` command creates a compile error. Initialize any local variables in advance.

**Note:** This command does not allow global variables to be declared; these variables must be declared in advance.

The `do` command works with multiple-line strings as well as single lines.

### Example

This statement performs the statement contained within quotation marks:

```
do "beep 2"  
do commandList[3]
```

## doneParsing()

### Syntax

```
parserObject.doneParsing()
```

### Description

Function; returns 1 (TRUE) when the parser has completed parsing a document using `parseURL()`. The return value is 0 (FALSE) until the parsing is complete.

### See also

`parseURL()`

## dot()

### Syntax

```
vector1.dot(vector2)
```

### Description

3D vector method; returns the sum of the products of the x, y, and z components of two vectors. If both vectors are normalized, the `dot` is the cosine of the angle between the two vectors.

To manually arrive at the dot of two vectors, multiply the x component of `vector1` by the x component of `vector2`, then multiply the y component of `vector1` by the y component of `vector2`, then multiply the z component of `vector1` by the z component of `vector2`, and finally add the three products together.

This method is identical to `dotProduct()` function.

### Example

In this example, the angle between the vectors `pos5` and `pos6` is 45 degrees. The `getNormalized` function returns the normalized values of `pos5` and `pos6`, and stores them in the variables `norm1` and `norm2`. The `dot` of `norm1` and `norm2` is 0.7071, which is the cosine of 45 degrees.

```
pos5 = vector(100, 100, 0)  
pos6 = vector(0, 100, 0)  
put pos5.angleBetween(pos6)  
-- 45.0000  
norm1 = pos5.getNormalized()  
put norm1  
-- vector( 0.7071, 0.7071, 0.0000 )  
norm2 = pos6.getNormalized()  
put norm2  
-- vector( 0.0000, 1.0000, 0.0000 )  
put norm1.dot(norm2)  
-- 0.7071
```

### See also

`dotProduct()`, `getNormalized`, `normalize`

## dotProduct()

### Syntax

```
vector1.dotProduct(vector2)
```

### Description

3D vector method; returns the sum of the products of the x, y, and z components of two vectors. If both vectors are normalized, the `dotproduct` is the cosine of the angle between the two vectors.

To manually arrive at the dot of two vectors, multiply the x component of `vector1` by the x component of `vector2`, then multiply the y component of `vector1` by the y component of `vector2`, then multiply the z component of `vector1` by the z component of `vector2`, and finally add the three products together.

This method is identical to `dot()` function.

### Example

In this example, the angle between the vectors `pos5` and `pos6` is 45°. The `getNormalized` function returns the normalized values of `pos5` and `pos6`, and stores them in the variables `norm1` and `norm2`. The `dotProduct` of `norm1` and `norm2` is 0.7071, which is the cosine of 45°.

```
pos5 = vector(100, 100, 0)
pos6 = vector(0, 100, 0)
put pos5.angleBetween(pos6)
-- 45.0000
norm1 = pos5.getNormalized()
put norm1
-- vector( 0.7071, 0.7071, 0.0000 )
norm2 = pos6.getNormalized()
put norm2
-- vector( 0.0000, 1.0000, 0.0000 )
put norm1.dotProduct(norm2)
-- 0.7071
```

### See also

`dot()`, `getNormalized`, `normalize`

## doubleClick

### Syntax

```
the doubleClick
```

### Description

Function; tests whether two mouse clicks within the time set for a double-click occurred as a double-click rather than two single clicks (TRUE), or if they didn't occur within the time set, treats them as single clicks (FALSE).

### Examples

This statement branches the playhead to the frame `Enter Bid` when the user double-clicks the mouse button:

```
if the doubleClick then go to frame "Enter Bid"
```

The following handler tests for a double-click. When the user clicks the mouse, a repeat loop runs for the time set for a double-click (20 ticks in this case). If a second click occurs within 20 ticks, the `doubleClickAction` handler runs. If a second click doesn't occur within the specified period, the `singleClickAction` handler runs:

```
on mouseUp
  if the doubleClick then exit
  startTimer
  repeat while the timer < 20
    if the mouseDown then
      doubleClickAction
      exit
    end if
  end repeat
  singleClickAction
end mouseUp
```

**See also**

`clickOn`, the `mouseDown` (system property), the `mouseUp` (system property)

## downloadNetThing

**Syntax**

```
downloadNetThing URL, localFile
```

**Description**

Command; copies a file from the Internet to a file on the local disk, while the current movie continues playing. Use `netDone` to find out whether downloading is finished.

- *URL*—The filename of any object that can be downloaded: for example, an FTP or HTTP server, an HTML page, an external cast member, a Director movie, or a graphic
- *localFile*—The pathname and filename for the file on the local disk

Director movies in authoring mode and projectors support the `downloadNetThing` command, but the Shockwave player does not. This protects users from unintentionally copying files from the Internet.

Although many network operations can be active at one time, running more than four concurrent operations usually slows down performance unacceptably.

Neither the Director movie's cache size nor the setting for the Check Documents option affects the behavior of the `downloadNetThing` command.

**Note:** Director for Java does not support the `downloadNetThing` command.

**Example**

These statements download an external cast member from a URL to the Director application folder and then make that file the external cast member named Cast of Thousands:

```
downloadNetThing("http://www.cbDeMille.com/Thousands.cst", the \
  applicationPath&"Thousands.Cst")
castLib("Cast of Thousands").fileName = the applicationPath&"Thousands.Cst"
```

**See also**

`importFileInto`, `netDone()`, `preloadNetThing()`

## drag

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).drag
```

### Description

3D *#particle* model resource property; indicates the percentage of each particle's velocity that is lost in each simulation step. This property has a range of 0 (no velocity lost) to 100 (all velocity lost and the particle stops moving). The default value is 0.

### Example

In this example, ThermoSystem is a model resource whose type is *#particle*. This statement sets the drag property of ThermoSystem to 5, applying a large resistance to the motion of the particles of ThermoSystem and preventing them from traveling very far.

```
member("Fires").modelResource("ThermoSystem").drag = 5
```

### See also

wind, gravity

## draw()

### Syntax

```
imageObject.draw(x1, y1, x2, y2, colorObjectOrParameterList)  
imageObject.draw(point(x, y), point(x, y), colorObjectOrParameterList)  
imageObject.draw(rect, colorObjectOrParameterList)
```

### Description

This function draws a line or an unfilled shape of color *colorObject* in a rectangular region of the given image object, as specified in any of the three ways shown. The draw returns a value of 1 if there is no error. You can use the optional property list *ParameterList* function to specify the following shape properties:

Property	Description
<i>#shapeType</i>	A symbol value of <i>#oval</i> , <i>#rect</i> , <i>#roundRect</i> , or <i>#line</i> . The default is <i>#line</i> .
<i>#lineSize</i>	The width of the line to use in drawing the shape.
<i>#color</i>	A color object, which determines the color of the shape border.

If you do not provide a parameter list, this function draws a 1-pixel line between the first and second points given or between the upper left and lower right corners of the given rectangle.

For best performance, with 8-bit or lower images the *colorObject* should contain an indexed color value. For 16- or 32-bit images, use an RGB color value.

If you want to fill a solid region, use the `fill()` function.

### Examples

This statement draws a 1-pixel, dark red, diagonal line from point (0, 0) to point (128, 86) within the image of member Happy.

```
member("Happy").image.draw(0, 0, 128, 86, rgb(150,0,0))
```



The following statement draws a dark red, 3-pixel unfilled oval within the image of member Happy. The oval is drawn within the rectangle (0, 0, 128, 86).

```
member("Happy").image.draw(0, 0, 128, 86, [#shapeType:#oval, #lineSize:3, \
#color: rgb(150, 0, 0)])
```

**See also**

`color()`, `copyPixels()`, `fill()`, `setPixel()`

## drawRect

**Syntax**

```
window windowName.drawRect
the drawRect of window windowName
```

**Description**

Window property; identifies the rectangular coordinates of the Stage of the movie that appears in the window. The coordinates are given as a rectangle, with entries in the order left, top, right, and bottom.

This property is useful for scaling or panning movies, but it does not rescale text and field cast members. Scaling bitmaps can affect performance.

This property can be tested and set.

**Example**

This statement displays the current coordinates of the movie window called Control Panel:

```
put the drawRect of window "Control Panel"
-- rect(10, 20, 200, 300).
```

The following statement sets the rectangle of the movie to the values of the rectangle named movieRectangle. The part of the movie within the rectangle is what appears in the window.

```
set the drawRect of window "Control Panel" to movieRectangle
```

The following lines cause the Stage to fill the main monitor area:

```
(the stage).drawRect = the desktopRectList[1]
(the stage).rect = the desktopRectList[1]
```

**See also**

`desktopRectList`, `rect (camera)`, `sourceRect`

## dropShadow

**Syntax**

```
member(whichCastMember).dropShadow
the dropShadow of member whichCastMember
```

**Description**

Cast member property; determines the size of the drop shadow in pixels, for text in a field cast member.

**Example**

This statement sets the drop shadow of the field cast member Comment to 5 pixels:

```
member("Comment").dropShadow = 5
```

## duplicate

### Syntax

```
vectorReference.duplicate()  
transformReference.duplicate()
```

### Description

3D vector and transform method; returns a copy of the vector or transform.

### Example

This statement creates a copy of the position of model 1 and stores it in the variable `zz`.

```
zz = member("MyRoom").model[1].transform.position.duplicate()
```

### See also

`clone`

## duplicateFrame

### Syntax

```
duplicateFrame
```

### Description

Command; duplicates the current frame and its content, inserts the duplicate frame after the current frame, and then makes the duplicate frame the current frame. This command can be used during Score generation only.

The `duplicateFrame` command performs the same function as the `insertFrame` command.

### Example

When used in the following handler, the `duplicateFrame` command creates a series of frames that have cast member `Ball` in the external cast `Toys` assigned to sprite channel 20. The number of frames is determined by the argument `numberOfFrames`.

```
on animBall numberOfFrames  
  beginRecording  
    sprite(20).member = member("Ball", "Toys")  
    repeat with i = 0 to numberOfFrames  
      duplicateFrame  
    end repeat  
  endRecording  
end
```

## duplicate() (list function)

### Syntax

```
(oldList).duplicate()  
duplicate(oldList)
```

### Description

List function; returns a copy of a list and copies nested lists (list items that also are lists) and their contents. The function is useful for saving a list's current content.

When you assign a list to a variable, the variable contains a reference to the list, not the list itself. This means any changes to the copy also affect the original list.

To see an example of `duplicate()` (list function) used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This statement makes a copy of the list `CustomersToday` and assigns it to the variable

```
CustomerRecord:
```

```
CustomerRecord = CustomersToday.duplicate()
```

## duplicate() (image function)

### Syntax

```
imageObject.duplicate()
```

### Description

This function creates and returns a copy of the given *imageObject*. The new image is completely independent of the original, and isn't linked to any cast member.

If you plan to make a lot of changes to an image, it's better to make a copy that's independent of a cast member.

### Example

This statement creates a new image object from the image of cast member `Lunar Surface` and places the new image object into the variable `workingImage`:

```
workingImage = member("Lunar Surface").image.duplicate()
```

### See also

`duplicate member`

## duplicate member

### Syntax

```
member(originalMember).duplicate()  
member(originalMember).duplicate({new})  
duplicate member original {, new}
```

### Description

Command; makes a copy of the cast member specified by *original*. The optional *new* parameter specifies a specific Cast window location for the duplicate cast member. If the *new* parameter isn't included, the duplicate cast member is placed in the first open Cast window position.

This command is best used during authoring rather than run time because it creates another cast member in memory, which could result in memory problems. Use the command during authoring if you want the changes to the cast to be permanently saved with the file.

### Examples

This statement makes a copy of cast member Desk and places it in the first empty Cast window position:

```
member("Desk").duplicate()
```

This statement makes a copy of cast member Desk and places it in the Cast window at position 125:

```
member("Desk").duplicate(125)
```

## duration

### Syntax

```
member(whichCastMember).duration  
the duration of member whichCastMember
```

### Description

Cast member property; determines the duration of the specified Shockwave Audio (SWA), transition, and QuickTime cast members.

- When *whichCastMember* is a streaming sound file, this property indicates the duration of the sound. The *duration* property returns 0 until streaming begins. Setting *preLoadTime* to 1 second allows the bit rate to return the actual duration.
- When *whichCastMember* is a digital video cast member, this property indicates the digital video's duration. The value is in ticks.
- When *whichCastMember* is a transition cast member, this property indicates the transition's duration. The value for the transition is in milliseconds. During playback, this setting has the same effect as the *Duration* setting in the Frame Transition dialog box.

This property can be tested for all cast members that support it, but only set for transitions.

To see an example of *duration* used in a completed movie, see the QT and Flash movie in the Learning/Lingo Examples folder inside the Director application folder.

## Examples

If the SWA cast member Louie Prima has been preloaded, this statement displays the sound's duration in the field cast member Duration Displayer:

```
on exitFrame
  if member("Louie Prima").state = 2 then
    member("Duration Displayer").text = member("Louie Prima").duration
  end if
end
```

You can use a behavior on a digital video sprite to loop the playhead in the current frame until the movie is finished playing, allowing it to continue when the end is reached:

```
property spriteNum

on exitFrame me
  myMember = sprite(spriteNum).member
  myDuration = member(myMember).duration
  myMovietime = sprite(spriteNum).movieTime
  if myDuration > myMovietime then
    go to the frame
  else
    go to the frame + 1
  end if
end
```

## duration (3D)

### Syntax

```
member(whichCastmember).motion(whichMotion).duration
motionObjectReference.duration
```

### Description

3D property; lets you get the time in millisecond that it takes the motion specified in the *whichMotion* parameter to play to completion. This property is always greater than or equal to 0.

### Example

This statement shows the length in milliseconds of the motion Kick.

```
put member("GbMember").motion("Kick").duration
-- 5100.0000
```

### See also

motion, currentTime (3D), play() (3D), queue() (3D)

## duration (RealMedia)

### Syntax

```
sprite(whichSprite).duration  
member(whichCastmember).duration
```

### Description

RealMedia sprite or cast member property; returns the duration of the RealMedia stream, in milliseconds. The duration of the RealMedia stream is not known until the cast member starts to play. If the stream is from a live feed or has not been played, the value of this property is 0. This property can be tested but not set.

### Examples

The following examples show that the duration of the RealMedia stream in sprite 2 and the cast member Real is 100,500 milliseconds (100.500 seconds).

```
put sprite(2).duration  
-- 100500  
  
put member("Real").duration  
-- 100500
```

### See also

play (RealMedia), seek, currentTime (RealMedia)

## editable

### Syntax

```
member(whichCastMember).editable  
the editable of member whichCastMember  
sprite(whichSprite).editable  
the editable of sprite whichSprite
```

### Description

Cast member and sprite property; determines whether the specified field cast member can be edited on the Stage (TRUE) or not (FALSE).

When the cast member property is set, the setting is applied to all sprites that contain the field. When the sprite property is set, only the specified sprite is affected.

You can also make a field cast member editable by using the Editable option in the Field Cast Member Properties dialog box.

You can make a field sprite editable by using the Editable option in the Score.

For the value set by Lingo to last beyond the current sprite, the sprite must be a puppet.

This property can be tested and set.

### Examples

This statement makes the field cast member Answer editable:

```
member("Answer").editable = TRUE
```

This handler first makes the sprite channel a puppet and then makes the field sprite editable:

```
on myNotes  
  puppetSprite 5, TRUE  
  sprite(5).editable = TRUE  
end
```

This statement checks whether a field sprite is editable and displays a message if it is:

```
if sprite(13).editable = TRUE then  
  member("Notice").text = "Please enter your answer below."  
end if
```

## editShortCutsEnabled

### Syntax

the editShortCutsEnabled

### Description

Movie property; determines whether cut, copy, and paste operations and their keyboard shortcuts function in the current movie. When set to `TRUE`, these text operations function. When set to `FALSE`, these operations are not allowed.

This property can be tested and set. The default is `TRUE` for movies made in Director 8 and later, `FALSE` for movies made in versions of Director prior to Director 8.

### Example

This statement disables cut, copy, and paste operations:

```
the editShortCutsEnabled = 0
```

## elapsedTime

### Syntax

sound(*channelNum*).elapsedTime  
the elapsedTime of sound *channelNum*

### Description

This read-only property gives the time, in milliseconds, that the current sound member in the given sound channel has been playing. It starts at 0 when the sound begins playing and increases as the sound plays, regardless of any looping, setting of the `currentTime` or other manipulation. Use the `currentTime` to test for the current absolute time within the sound.

The value of this property is a floating-point number, allowing for measurement of sound playback to fractional milliseconds.

### Example

This idle handler displays the elapsed time for sound channel 4 in a field on the Stage during idles:

```
on idle  
    member("time").text = string(sound(4).elapsedTime)  
end idle
```

### See also

`currentTime`, `loopCount`, `loopsRemaining`, `rewind()`

## emissive

### Syntax

```
member(whichCastmember).shader(whichShader).emissive  
member(whichCastmember).model(whichModel).shader.emissive  
member(whichCastmember).model(whichModel).shaderList[[index]].\emissive
```

### Description

3D #standard shader property; adds light to the shader independently of the lighting in the scene. For example, a model using a shader whose `emissive` property is set to `rgb(255, 255, 255)` will appear to be illuminated by a white light, even if there are no lights in the scene. The model will not, however, illuminate any other models or contribute any light to the scene.



The default value for this property is `rgb(0, 0, 0)`.

### Example

This statement sets the `emissive` property of the shader named `Globe` to `rgb(255, 0, 0)`. Models using this shader will appear to be illuminated by a red light:

```
member("MysteryWorld").shader("Globe").emissive = rgb(255, 0, 0)
```

### See also

`silhouettes`

## emitter

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
    emitter.numParticles
member(whichCastmember).modelResource(whichModelResource).\
    emitter.mode
member(whichCastmember).modelResource(whichModelResource).\
    emitter.loop
member(whichCastmember).modelResource(whichModelResource).\
    emitter.direction
member(whichCastmember).modelResource(whichModelResource).\
    emitter.region
member(whichCastmember).modelResource(whichModelResource).\
    emitter.distribution
member(whichCastmember).modelResource(whichModelResource).\
    emitter.angle
member(whichCastmember).modelResource(whichModelResource).\
    emitter.path
member(whichCastmember).modelResource(whichModelResource).\
    emitter.pathStrength
member(whichCastmember).modelResource(whichModelResource).\
    emitter.minSpeed
member(whichCastmember).modelResource(whichModelResource).\
    emitter.maxSpeed
```

### Description

3D particle system element; controls the initial propulsion of particles from a model resource whose type is `#particle`.

The “See also” section of this entry contains a complete list of emitter properties. For more information, see the individual property entries.

### See also

`numParticles`, `loop` (`emitter`), `direction`, `distribution`, `region`, `angle`, `path`, `pathStrength`, `minSpeed`, `maxSpeed`

## EMPTY

### Syntax

EMPTY

### Description

Character constant; represents the empty string, "", a string with no characters.

### Example

This statement erases all characters in the field cast member Notice by setting the field to EMPTY:

```
member("Notice").text = EMPTY
```

## emulateMultiButtonMouse

### Syntax

the emulateMultiButtonMouse

### Description

System property; determines whether a movie interprets a mouse click with the Control key pressed on the Macintosh the same as a right mouse click in Windows (TRUE) or not (FALSE).

Right-clicking has no direct Macintosh equivalent.

Setting this property to TRUE lets you provide consistent mouse button responses for cross-platform movies.

### Example

The following statement checks if the computer is a Macintosh and if so, sets the emulateMultiButtonMouse property to TRUE:

```
if the platform contains "Macintosh" then the emulateMultiButtonMouse = TRUE
```

### See also

keyPressed(), rightMouseDown (system property), rightMouseUp (system property)

## enabled

### Syntax

the enabled of menuItem *whichItem* of menu *whichMenu*

### Description

Menu item property; determines whether the menu item specified by *whichItem* is displayed in plain text and is selectable (TRUE, default) or appears dimmed and is not selectable (FALSE).

The expression *whichItem* can be either a menu item name or a menu item number. The expression *whichMenu* can be either a menu name or a menu number.

The enabled property can be tested and set.

**Note:** Menus are not available in Shockwave.

### Example

This handler enables or disables all the items in the specified menu. The argument `theMenu` specifies the menu; the argument `Setting` specifies `TRUE` or `FALSE`. For example, the calling statement `ableMenu ("Special", FALSE)` disables all the items in the Special menu.

```
on ableMenu theMenu, vSetting
  set n = the number of menuItems of menu theMenu
  repeat with i = 1 to n
    set the enabled of menuItem i of menu theMenu to vSetting
  end repeat
end ableMenu
```

### See also

`name (menu property)`, `number (menus)`, `checkMark`, `script`, `number (menu items)`

## enabled (collision)

### Syntax

`member(whichCastmember).model(whichModel).collision.enabled`

### Description

3D collision property; allows you to get or set whether (`TRUE`) or not (`FALSE`) collisions are detected on models. Setting this property to `FALSE` temporarily disables the collision modifier without removing it from the model.

The default setting for this property is `TRUE`.

### Example

This statement activates the collision modifier for the model box:

```
member("3d world").model("box").collision.enabled = TRUE
```

### See also

`addModifier`, `collision (modifier)`, `modifier`

## enabled (fog)

### Syntax

`member(whichCastmember).camera(whichCamera).fog.enabled`  
`sprite(whichSprite).camera{( index )}.fog.enabled`

### Description

3D camera property; indicates whether the camera adds fog to the view from the camera. The default setting for this property is `FALSE`.

### Example

This statement creates fog in the view from the camera named BayView:

```
member("MyYard").camera("BayView").fog.enabled = TRUE
```

### See also

`fog`

## enabled (sds)

### Syntax

```
member(whichCastmember).model(whichModel).sds.enabled
```

### Description

3D sds modifier property; indicates whether the sds modifier attached to a model is used by the model.

The default setting for this property is TRUE.

An attempt to add the sds modifier to a model that already has the `inker` or `toon` modifier attached fails without an error message. Likewise, an attempt to add the `inker` or `toon` modifier to a model that already has the sds modifier attached also fails without an error message. Be careful when using the sds modifier with the `lod` modifier. For more information, see the `sds (modifier)` entry.

### Example

This statement turns on the sds modifier attached to the model Baby:

```
member("Scene").model("Baby").sds.enabled = TRUE
```

### See also

`sds (modifier)`, `modifier`, `addModifier`

## enableHotSpot

### Syntax

```
enableHotSpot(sprite whichQTVRSprite, hotSpotID, trueOrFalse)
```

### Description

QTVR (QuickTime VR) command; determines whether the specified hot spot for the specified QTVR sprite is enabled (TRUE), or disabled (FALSE).

## end

### Syntax

```
end
```

### Description

Keyword; marks the end of handlers and multiple-line control structures.

## end case

### Syntax

```
end case
```

### Description

Keyword; ends a case statement.

### Example

This handler uses the `end case` keyword to end the case statement:

```
on keyDown
  case the key
    of "A": go to frame "Apple"
    of "B", "C" :
      puppetTransition 99
      go to frame "Mango"
    otherwise beep
  end case
end keyDown
```

### See also

case

## endAngle

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
  endAngle
```

### Description

3D `#cylinder` or `#sphere` model resource property; indicates how much of the sphere or cylinder is drawn.

The surface of a sphere is generated by sweeping a 2D half circle arc around the sphere's Y axis from `startAngle` to `endAngle`. If `startAngle` is set to 0 and `endAngle` is set to 360, the result is a complete sphere. To draw a section of a sphere, set `endAngle` to a value less than 360.

The surface of a cylinder is generated by sweeping a 2D line around the sphere's Y axis from `startAngle` to `endAngle`. If `startAngle` is set to 0 and `endAngle` is set to 360, the result is a complete cylinder. To draw a section of a cylinder, set `endAngle` to a value less than 360.

The default setting for this property is 360.

### Example

For this example, assume that the cast member named `MyMember` contains a model that uses the model resource named `Sphere4`, whose `endAngle` value is 310, leaving an opening of 50°. The handler `closeSphere` closes that opening in a way that makes it look like it is sliding shut. The repeat loop changes the `endAngle` value of the sphere 1° at a time. The `updateStage` command in the repeat loop forces the Stage to redraw after every 1° increment.

```
on closeSphere
  MyAngle = member("MyMember").modelResource("Sphere4").endAngle
  repeat with r = 1 to 50
    MyAngle = MyAngle + 1
    member("MyMember").modelResource("Sphere4").endAngle = MyAngle
    updateStage
  end repeat
end
```

### See also

state (3D)

## endColor

### Syntax

the endColor of member *whichCastMember*

### Description

Vector shape cast member property; the ending color of a gradient shape's fill specified as an RGB value.

endColor is only valid when the fillMode is set to #gradient, and the starting color is set with fillColor.

This property can be tested and set.

To see an example of endColor used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

### See also

color(), fillColor, fillMode

## endFrame()

### Syntax

sprite(*whichSprite*).endFrame

### Description

Function; returns the frame number of the end frame of the sprite span.

This function is useful in determining the span in the Score of a particular sprite. This function is available only in a frame that contains the sprite. It cannot be applied to sprites in different frames of the movie, nor is it possible to set this value.

### Example

This statement output reports the ending frame of the sprite in channel 5 in the Message window:

```
put sprite(5).endFrame
```

### See also

startFrame

## endRecording

### Syntax

endRecording

### Description

Keyword; ends a Score update session. You can resume control of Score channels through puppeting after the endRecording keyword is issued.

### Example

When used in the following handler, the `endRecording` keyword ends the Score generation session:

```
on animBall numberOfFrames
  beginRecording
    horizontal = 0
    vertical = 100
    repeat with i = 1 to numberOfFrames
      go to frame i
      sprite(20).member = member "Ball"
      sprite(20).locH = horizontal
      sprite(20).locV = vertical
      horizontal = horizontal + 3
      vertical = vertical + 2
      updateFrame
    end repeat
  endRecording
end
```

### See also

`beginRecording`, `scriptNum`, `tweened`, `updateFrame`

## end repeat

### See

`repeat while`, `repeat with`, `repeat with...in list`, `repeat with...down to`

## on endSprite

### Syntax

```
on endSprite
  statement(s)
end
```

### Description

System message and event handler; contains Lingo that runs when the playhead leaves a sprite and goes to a frame in which the sprite doesn't exist. It is generated after `exitFrame`.

Place `on endSprite` handlers in a behavior script.

Director destroys instances of any behavior scripts attached to the sprite immediately after the `endSprite` event occurs.

The event handler is passed the behavior or frame script reference `me` if used in a behavior. This `endSprite` message is sent after the `exitFrame` message if the playhead plays to the end of the frame.

The `go`, `play`, and `updateStage` commands are disabled in an `on endSprite` handler.

### Example

This handler runs when the playhead exits a sprite:

```
on endSprite me
  -- clean up
  gNumberOfSharks = gNumberOfSharks - 1
  puppetSound(5,0)
end
```

### See also

`on beginSprite`, `on exitFrame`

## endTellTarget()

See `tellTarget()`.

## endTime

### Syntax

`sound(channelNum).endTime`  
the `endTime` of sound `channelNum`

### Description

This property is the specified end time of the currently playing, paused or queued sound. This is the time within the sound member when it will stop playing. It's a floating-point value, allowing for measurement and control of sound playback to fractions of milliseconds. The default value is the normal end of the sound.

This property may be set to a value other than the normal end of the sound only when passed as a parameter with the `queue()` or `setPlaylist()` commands.

### Example

This Lingo checks whether the sound member `Jingle` is set to play all the way through in sound channel 1:

```
if sound(1).startTime > 0 and sound(1).endTime < member("Jingle").duration
    then
        alert "Not playing the whole sound"
    end if
```

### See also

`setPlaylist()`, `queue()`, `startTime`

## ENTER

### Syntax

`ENTER`

### Description

Character constant; represents the Enter key (Windows) or the Return key (Macintosh) for a carriage return.

On PC keyboards, the element `ENTER` refers only to the Enter key on the numeric keypad.

For a movie that plays back as an applet, use `RETURN` to specify both the Return key in Windows and the Enter key on the Macintosh.

### Example

This statement checks whether the Enter key is pressed and if it is, sends the playhead to the frame `addSum`:

```
on keyDown
    if the key = ENTER then go to frame "addSum"
end
```

### See also

`RETURN` (constant)



## on enterFrame

### Syntax

```
on enterFrame
    statement(s)
end
```

### Description

System message and event handler; contains statements that run each time the playhead enters the frame.

Place `on enterFrame` handlers in behavior, frame, or movie scripts, as follows:

- To assign the handler to an individual sprite, put the handler in a behavior attached to the sprite.
- To assign the handler to an individual frame, put the handler in the frame script.
- To assign the handler to every frame (unless you explicitly instruct the movie otherwise), put the `on enterFrame` handler in a movie script. The handler executes every time the playhead enters a frame unless the frame script has its own handler. If the frame script has its own handler, the `on enterFrame` handler in the frame script overrides the `on enterFrame` handler in the movie script.

The order of frame events is `stepFrame`, `prepareFrame`, `enterFrame`, and `exitFrame`.

This event is passed the object reference `me` if used in a behavior.

### Example

This handler turns off the puppet condition for sprites 1 through 5 each time the playhead enters the frame:

```
on enterFrame
    repeat with i = 1 to 5
        puppetSprite i, FALSE
    end repeat
end
```

## environment

### Syntax

```
the environment
the environment.propertyName
```

### Description

System property; this property contains a list with information about the environment under which the Director content is currently running.

This design enables Macromedia to add information to the `environment` property in the future, without affecting existing movies.

The information is in the form of property and value pairs for that area.

---

#shockMachine	Integer TRUE or FALSE value indicating whether the movie is playing in ShockMachine.
#shockMachineVersion	String indicating the installed version number of ShockMachine.
#platform	String containing "Macintosh,PowerPC", or "Windows,32". This is based on the current OS and hardware that the movie is running under.
#runMode	String containing "Author", "Projector", "Plugin", or "Java Applet". This is based on the current application that the movie is running under.
#colorDepth	Integer representing the bit depth of the monitor the Stage appears on. Possible values are 1, 2, 4, 8, 16, or 32.
#internetConnected	Symbol indicating whether the computer the movie is playing on has an active Internet connection. Possible values are <i>#online</i> and <i>#offline</i> .
#uiLanguage	String indicating the language the computer is using to display its user interface. This can be different from the <i>#osLanguage</i> on computers with specific language kits installed.
#osLanguage	String indicating the native language of the computer's operating system.
#productBuildVersion	String indicating the internal build number of the playback application.

---

The properties contain exactly the same information as the properties and functions of the same name.

### Example

This statement displays the environment list in the Message window:

```
put the environment
-- [#shockMachine: 0, #shockMachineVersion: "", #platform:
   "Macintosh,PowerPC", #runMode: "Author", #colorDepth: 32,
   #internetConnected: #online, #uiLanguage: "English", #osLanguage: "English",
   #productBuildVersion: "151"]
```

### See also

*colorDepth*, *platform*, *runMode*

## erase member

### Syntax

```
member(whichCastMember).erase()
erase member whichCastMember
```

### Description

Command; deletes the specified cast member and leaves its slot in the Cast window empty.

For best results, use this command during authoring and not in projectors, which can cause memory problems.

### Examples

This statement deletes the cast member named Gear in the Hardware cast:

```
member("Gear", "Hardware").erase()
```

This handler deletes cast members start through finish:

```
on deleteMember start, finish
  repeat with i = start to finish
    member(i).erase()
  end repeat
end on deleteMember
```

**See also**

`new()`

## error

**Syntax**

```
member(whichCastmember).model(whichModel).sds.error
```

**Description**

3D `#sds` modifier property; indicates the percentage of error tolerated by the modifier when synthesizing geometric detail in models.

This property works only when the modifier's `subdivision` property is set to `#adaptive`. The `tension` and `depth` (3D) properties of the modifier combine with the `error` property to control the amount of subdivision performed by the modifier.

**Example**

The following statement sets the `error` property of the `#sds` modifier of the model named `Baby` to 0. If the modifier's `tension` setting is low, its `depth` setting is high, and its `subdivision` setting is `#adaptive`, this will cause a very pronounced effect on `Baby`'s geometry.

```
member("Scene").model("Baby").sds.error = 0
```

**See also**

`sds` (modifier), `subdivision`, `depth` (3D), `tension`

## on EvalScript

**Syntax**

```
on EvalScript aParam
  statement(s)
end
```

**Description**

System message and event handler; in a Shockwave movie, contains statements that run when the handler receives an `EvalScript` message from a browser. The parameter is a string passed in from the browser.

- The `EvalScript` message can include a string that Director can interpret as a Lingo statement. Lingo cannot accept nested strings. If the handler you are calling expects a string as a parameter, pass the parameter as a symbol.
- The `on EvalScript` handler is called by the `EvalScript()` scripting method from JavaScript or VBScript in a browser.

The Director player for Java doesn't support the `on EvalScript` handler. To enable communication between an applet and a browser, use Java, JavaScript, or VBScript.

Include only those behaviors in `on EvalScript` that you want users to control; for security reasons, don't give complete access to behaviors.

**Note:** If you place a `return` at the end of your `EvalScript` handler, the value returned can be used by JavaScript in the browser.

### Examples

This shows how to make the playhead jump to a specific frame depending on what frame is passed in as the parameter:

```
on EvalScript aParam
  go frame aParam
end
```

This handler runs the statement `go frame aParam` if it receives an `EvalScript` message that includes `dog`, `cat`, or `tree` as an argument:

```
on EvalScript aParam
  case aParam of
    "dog", "cat", "tree": go frame aParam
  end case
end
```

A possible calling statement for this in JavaScript would be `EvalScript ("dog")`.

This handler takes an argument that can be a number or symbol:

```
on EvalScript aParam
  if word 1 of aParam = "myHandler" then
    do aParam
  end if
end
```

The following handler normally requires a string as its argument. The argument is received as a symbol and then converted to a string within the handler by the `string` function:

```
on myHandler aParam
  go to frame string(aParam)
end
```

### See also

`externalEvent`, `return` (keyword)

## eventPassMode

### Syntax

`sprite(whichFlashSprite).eventPassMode`  
the `eventPassMode` of `sprite` *whichFlashSprite*  
`member(whichFlashMember).eventPassMode`  
the `eventPassMode` of `member` *whichFlashMember*

### Description

Flash cast member property and sprite property; controls when a Flash movie passes mouse events to behaviors that are attached to sprites that lie underneath the flash sprite. The `eventPassMode` property can have these values:

- `#passAlways` (default)—Always passes mouse events.
- `#passButton`—Passes mouse events only when a button in the Flash movie is clicked.
- `#passNotButton`—Passes mouse events only when a nonbutton object is clicked.
- `#passNever`—Never passes mouse events.

This property can be tested and set.

### Example

The following frame script checks to see whether the buttons in a Flash movie sprite are currently enabled, and if so, sets `eventPassMode` to `#passNotButton`; if the buttons are disabled, the script sets `eventPassMode` to `#passAlways`. The effect of this script is the following:

- Mouse events on nonbutton objects always pass to sprite scripts.
- Mouse events on button objects are passed to sprite scripts when the buttons are disabled. When the buttons are enabled, mouse events on buttons are stopped.

```
on enterFrame
  if sprite(5).buttonsEnabled = TRUE then
    sprite(5).eventPassMode= #passNotButton
  else
    sprite(5).eventPassMode = #passAlways
  end if
end
```

## exit

### Syntax

```
exit
```

### Description

Keyword; instructs Lingo to leave a handler and return to where the handler was called. If the handler is nested within another handler, Lingo returns to the main handler.

### Example

The first statement of this script checks whether the monitor is set to black and white and then exits if it is:

```
on setColors
  if the colorDepth = 1 then exit
  sprite(1).foreColor = 35
end
```

### See also

abort, halt, quit, pass, return (keyword)

## on exitFrame

### Syntax

```
on exitFrame
  statement(s)
end
```

### Description

System message and event handler; contains statements that run each time the playhead exits the frame that the `on exitFrame` handler is attached to. The `on exitFrame` handler is a useful place for Lingo that resets conditions that are no longer appropriate after leaving the frame.

Place `on exitFrame` handlers in behavior, frame, or movie scripts, as follows:

- To assign the handler to an individual sprite, put the handler in a behavior attached to the sprite.
- To assign the handler to an individual frame, put the handler in the frame script.
- To assign the handler to every frame unless explicitly instructed otherwise, put the handler in a movie script. The `on exitFrame` handler then executes every time the playhead exits the frame unless the frame script has its own `on exitFrame` handler. When the frame script has its own `on exitFrame` handler, the `on exitFrame` handler in the frame script overrides the one in the movie script.

This event is passed the sprite script or frame script reference `me` if it is used in a behavior. The order of frame events is `prepareFrame`, `enterFrame`, and `exitFrame`.

### Examples

This handler turns off all puppet conditions when the playhead exits the frame:

```
on exitFrame me
  repeat with i = 48 down to 1
    sprite(i).puppet = FALSE
  end repeat
end
```

This handler branches the playhead to a specified frame if the value in the global variable `vTotal` exceeds 1000 when the playhead exits the frame:

```
global vTotal

on exitFrame
  if vTotal > 1000 then go to frame "Finished"
end
```

### See also

`on enterFrame`

## exitLock

### Syntax

```
the exitLock
```

### Description

Movie property; determines whether a user can quit to the Windows desktop or Macintosh Finder from projectors (`FALSE`, default) or not (`TRUE`).

The user can quit to the desktop by pressing `Control+period` (Windows) or `Command+period` (Macintosh), `Control+Q` (Windows) or `Command+Q` (Macintosh), or `Control+W` (Windows) or `Command+W` (Macintosh); the `Escape` key is also supported in Windows.

This property can be tested and set.

### Examples

This statement sets the `exitLock` property to `TRUE`:

```
set the exitLock to TRUE
```

Assuming that `exitLock` is set to `TRUE`, nothing occurs automatically when the `Control+period/Q/W`, `Esc`, or `Command+period/Q/W` keys are used. This handler checks keyboard input for keys to exit and takes the user to a custom quit sequence:

```
on checkExit
  if the commandDown and (the key = "." or the key = "q") and the exitLock =
    TRUE then go to frame "quit sequence"
end
```

## exit repeat

### Syntax

```
exit repeat
```

### Description

**Keyword;** instructs Lingo to leave a repeat loop and go to the statement following the `end repeat` statement but to remain within the current handler or method.

The `exit repeat` keyword is useful for breaking out of a repeat loop when a specified condition—such as two values being equal or a variable being a certain value—exists.

### Example

The following handler searches for the position of the first vowel in a string represented by the variable `testString`. As soon as the first vowel is found, the `exit repeat` command instructs Lingo to leave the repeat loop and go to the statement `return i`:

```
"on findVowel testString
  repeat with i = 1 to testString.char[testString.char.count]
    if "aeiou" contains testString.char[i] then exit repeat
  end repeat
  return i
end"
```

### See also

`repeat while`, `repeat with`

## exp()

### Syntax

```
(integerOrFloat).exp
exp(integerOrFloat)
```

### Description

**Function;** calculates *e*, the natural logarithm base, to the power specified by *integerOrFloat*.

### Example

The following statement calculates the value of *e* to the exponent 5:

```
put (5).exp
-- 148.4132
```

## externalEvent

### Syntax

`externalEvent "string"`

### Description

Command; sends a string to the browser that the browser can interpret as a scripting language instruction, allowing a movie playing or a browser to communicate with the HTML page in which it is embedded. The string sent by `externalEvent` must be in a scripting language supported by the browser.

This command works only for movies in browsers. To enable communication between an applet and a browser, use Java, JavaScript, or VBScript.

**Note:** The `externalEvent` command does not produce a return value. There is no immediate way to determine whether the browser handled the event or ignored it. Use `on EvalScript` within the browser to return a message to the movie.

### Examples

The following statements use `externalEvent` in the LiveConnect scripting environment, which is supported by Netscape 3.x and later.

LiveConnect evaluates the string passed by `externalEvent` as a function call. JavaScript authors must define and name this function in the HTML header. In the movie, the function name and parameters are defined as a string in `externalEvent`. Because the parameters must be interpreted by the browser as separate strings, each parameter is surrounded by single quotation marks.

Statements within HTML:

```
function MyFunction(parm1, parm2) {  
    //script here  
}
```

Statements within a script in the movie:

```
externalEvent ("MyFunction('parm1','parm2')")
```

The following statements use `externalEvent` in the ActiveX scripting environment used by Internet Explorer in Windows. ActiveX treats `externalEvent` as an event and processes this event and its string parameter the same as an `onClick` event in a button object.

- Statements within HTML:

In the HTML header, define a function to catch the event; this example is in VBScript:

```
Sub  
NameOfShockwaveInstance_externalEvent(aParam)  
    'script here  
End Sub
```

Alternatively, define a script for the event:

```
<SCRIPT FOR="NameOfShockwaveInstance"  
EVENT="externalEvent(aParam)"  
LANGUAGE="VBScript">  
    'script here  
</SCRIPT>
```

Within the movie, include the function and any parameters as part of the string for `externalEvent`:

```
externalEvent ("MyFunction ('parm1','parm2')")
```

### See also

`on EvalScript`



## externalParamCount()

### Syntax

```
externalParamCount()
```

### Description

Function; returns the number of parameters that an HTML <EMBED> or <OBJECT> tag is passing to a Shockwave movie.

This function is valid only for Shockwave movies that are running in a browser. It doesn't work for movies during authoring or for projectors.

### Example

This handler determines whether an <OBJECT> or <EMBED> tag is passing any external parameters to a Shockwave movie and runs Lingo statements if parameters are being passed:

```
if externalParamCount() > 0 then
    -- perform some action
end if
```

### See also

```
externalParamName(), externalParamValue()
```

## externalParamName()

### Syntax

```
externalParamName(n)
```

### Description

Function; returns the name of a specific parameter in the list of external parameters from an HTML <EMBED> or <OBJECT> tag. This function is valid only for Shockwave movies that are running in a browser. It cannot be used with Director movies or projectors.

- If *n* is an integer, `externalParamName` returns the *n*th parameter name in the list.
- If *n* is a string, `externalParamName` returns *n* if any of the external parameter names matches *n*. The match is not case sensitive. If no matching parameter name is found, `externalParamName` returns `VOID`.

### Example

This statement places the value of a given external parameter in the variable `myVariable`:

```
if externalParamName ("swURLString") = "swURLString" then
    myVariable = externalParamValue ("swURLString")
end if
```

### See also

```
externalParamCount(), externalParamValue()
```

## externalParamValue()

### Syntax

`externalParamValue(n)`

### Description

Function; returns a specific value from the external parameter list in an HTML <EMBED> or <OBJECT> tag. This function is valid only for Shockwave movies that are running in a browser. It can't be used with movies running in the authoring environment or projectors.

- If *n* is an integer, `externalParamValue` returns the *n*th parameter value from the external parameter list.
- If *n* is a string, `externalParamValue` returns the value associated with the first name that matches *n*. The match isn't case sensitive. If no such parameter value exists, `externalParamValue` returns VOID.

This function's behavior in an applet differs from that in other Director movies. In an applet, `externalParamValue` does the following:

- Returns the applet's parameters instead of the <EMBED> tag parameters.
- Accepts only string parameters.
- Returns a zero-length string rather than VOID.

See “Parameters for OBJECT and EMBED tags” and “Parameters accessible from Lingo” on the Director Developers Center Web site.

### Example

This statement places the value of an external parameter in the variable `myVariable`:

```
if externalParamName ("swURLString") = "swURLString" then
    myVariable = externalParamValue ("swURLString")
end if
```

### See also

`externalParamCount()`, `externalParamName()`

## extractAlpha()

### Syntax

`imageObject.extractAlpha()`

### Description

This function copies the alpha channel from the given 32-bit image and returns it as a new image object. The result is an 8-bit grayscale image representing the alpha channel.

This function is useful for downsampling 32-bit images with alpha channels.

### Example

This statement places the alpha channel of the image of member 1 into the variable `mainAlpha`:

```
mainAlpha = member(1).image.extractAlpha()
setAlpha(), useAlpha
```

# extrude3D

## Syntax

`member(whichTextCastmember).extrude3D(member(which3dCastmember))`

## Description

3D command; creates a new `#extruder` model resource in the 3D cast member *which3dCastmember* from the text in *whichTextCastmember*.

Note that this is not the same as using the 3D `displayMode` property of a text cast member.

## To create a model using extrude3D:

- 1 Create a new `#extruder` model resource in a 3D cast member:

```
textResource = member("textMember").extrude3D(member\  
("3DMember"))
```

- 2 Create a new model using the model resource created in step 1:

```
member("3DMember").newModel("myText", textResource)
```

## Example

In this example, Logo is a text cast member and Scene is a 3D cast member. The first line creates a model resource in Scene which is a 3D version of the text in Logo. The second line uses this model resource to create a model named 3dLogo.

```
myTextModelResource = member("Logo").extrude3d(member("Scene"))  
member("Scene").newModel("3dLogo", myTextModelResource)
```

## See also

`bevelDepth`, `bevelType`, `displayFace`, `smoothness`, `tunnelDepth`, `displayMode`

# face

## Syntax

```
member(whichCastmember).modelResource(whichModelResource).\  
    face.count  
member(whichCastmember).modelResource(whichModelResource).\  
    face[index].colors  
member(whichCastmember).modelResource(whichModelResource).\  
    face[index].normals  
member(whichCastmember).modelResource(whichModelResource).\  
    face[index].shader  
member(whichCastmember).modelResource(whichModelResource).\  
    face[index].textureCoordinates  
member(whichCastmember).modelResource(whichModelResource).\  
    face[index].vertices  
member(whichCastmember).model(whichModel).meshdeform.\  
    face.count  
member(whichCastmember).model(whichModel).meshdeform.\  
    mesh[index].face.count  
member(whichCastmember).model(whichModel).meshdeform.\  
    mesh[meshIndex].face[faceIndex]  
member(whichCastmember).model(whichModel).meshdeform.\  
    mesh[meshIndex].face[faceIndex].neighbor{[neighborIndex]}
```

### Description

3D `#mesh` model resource and `meshdeform` modifier property. All model resources are meshes composed of triangles. Each triangle is a face.

You can access the properties of the faces of model resources whose type is `#mesh`. Changes to any of these properties do not take effect until you call the `build()` command.

**Note:** For detailed information about the following properties, see the individual property entries.

- `count` indicates the number of triangles in the mesh.
- `colors` indicates which indices in the color list of the model resource to use for each of the vertices of the face.
- `normals` indicates which indices in the normal list of the model resource to use for each of the vertices of the face.
- `shadowPercentage` identifies the shader used when the face is rendered.
- `textureCoordinates` indicates which indices in the texture coordinate list of the model resource to use for each of the vertices of the face.
- `vertices` indicates which indices in the vertex list of the model resource to use to define the face.

See the entry for `meshDeform` for descriptions of its face properties.

### See also

`build()`, `newMesh`, `meshDeform` (modifier)

## face[ ]

### Syntax

```
member(whichCastmember).model(whichModel).meshdeform.\  
mesh[meshIndex].face[faceIndex]
```

### Description

3D `meshdeform` modifier property; indicates which indices in the vertex list of the model resource were used to define the face.

This property can be tested but not set. You can specify the vertices of a face of the `#mesh` model resource by setting its `vertexList` and `vertices` properties and calling the `build` command.

### Example

This statement shows that the first face of the first mesh of the model named `Floor` is defined by the first three vectors in the vertex list of the model resource used by `Floor`:

```
put member("Scene").model("Floor").meshdeform.mesh[1].face[1]  
-- [1, 2, 3]
```

### See also

`meshDeform` (modifier), `face`, `vertexList` (mesh deform), `vertices`

## fadeIn()

### Syntax

```
sound(channelNum).fadeIn({milliseconds})  
fadeIn(sound(channelNum) {, milliseconds })
```

### Description

This function immediately sets the volume of sound channel *channelNum* to zero and then brings it back to the current volume over the given number of milliseconds. The default is 1000 milliseconds (1 second) value is given.

The current pan setting is retained for the entire fade.

### Example

This Lingo fades in sound channel 3 over a period of 3 seconds from the beginning of cast member *introMusic2*:

```
sound(3).play(member("introMusic2"))  
sound(3).fadeIn(3000)
```

### See also

[fadeOut\(\)](#), [fadeTo\(\)](#), [pan](#) (sound property), [volume](#) (sound channel)

## fadeOut()

### Syntax

```
sound(channelNum).fadeOut({milliseconds})  
fadeOut(sound(channelNum) {, milliseconds })
```

### Description

This function gradually reduces the volume of sound channel *channelNum* to zero over the given number of milliseconds, or 1000 milliseconds (1 second) if no value is given.

The current pan setting is retained for the entire fade.

### Example

This statement fades out sound channel 3 over a period of 5 seconds:

```
sound(3).fadeOut(5000)
```

### See also

[fadeIn\(\)](#), [fadeTo\(\)](#), [pan](#) (sound property), [volume](#) (sound channel)

## fadeTo()

### Syntax

```
sound(channelNum).fadeTo(volume {, milliseconds })  
fadeTo(sound(channelNum), volume {, milliseconds })
```

### Description

This function gradually changes the volume of sound channel *channelNum* to the specified volume over the given number of milliseconds, or 1000 milliseconds (1 second) if no value is given. The range of values for volume is 0-255.

The current pan setting is retained for the entire fade.

To see an example of `fadeTo()` used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

#### Example

The following statement changes the volume of sound channel 4 to 150 over a period of 2 seconds. It can be a fade up or a fade down, depending on the original volume of sound channel 4 when the fade begins.

```
sound(4).fadeTo(150, 2000)
```

#### See also

`fadeIn()`, `fadeOut()`, `pan (sound property)`, `volume (sound channel)`

## FALSE

#### Syntax

`FALSE`

#### Description

Constant; applies to an expression that is logically `FALSE`, such as `2 > 3`. When treated as a number value, `FALSE` has the numerical value of 0. Conversely, 0 is treated as `FALSE`.

#### Example

This statement turns off the `soundEnabled` property by setting it to `FALSE`:

```
the soundEnabled = FALSE
```

#### See also

`if`, `not`, `TRUE`

## far (fog)

#### Syntax

```
member(whichCastmember).camera(whichCamera).fog.far  
sprite(whichSprite).camera{( index )}.fog.far
```

#### Description

3D camera property; indicates the distance from the camera, in world units, where the fog reaches its maximum density when the camera's `fog.enabled` property is set to `TRUE`.

The default value for this property is 1000.

#### Example

The following statement sets the `far` property of the fog of the camera named `BayView` to 5000. If the fog's `enabled` property is set to `TRUE`, the fog will be densest 5000 world units in front of the camera.

```
member("MyYard").camera("BayView").fog.far = 5000
```

#### See also

`fog`, `near (fog)`

## field

### Syntax

`field` *whichField*

### Description

Keyword; refers to the field cast member specified by *whichField*.

- When *whichField* is a string, it is used as the cast member name.
- When *whichField* is an integer, it is used as the cast member number.

Character strings and chunk expressions can be read from or placed in the field.

The term `field` was used in earlier versions of Director and is maintained for backward compatibility. For new movies, use `member` to refer to field cast members.

### Examples

This statement places the characters 5 through 10 of the field name entry in the variable `myKeyword`:

```
myKeyword = field("entry").char[5..10]
```

This statement checks whether the user entered the word *desk* and, if so, goes to the frame `deskBid`:

```
if member "bid" contains "desk" then go to "deskBid"
```

### See also

`char...of`, `item...of`, `line...of`, `word...of`

## fieldOfView

### Syntax

`sprite(whichQTVRSprite).fieldOfView`  
the `fieldOfView` of sprite *whichQTVRSprite*

### Description

QTVR sprite property; gives the specified sprite's current field of view in degrees.

This property can be tested and set.

## fieldOfView (3D)

### Syntax

`member(whichCastmember).camera(whichCamera).fieldOfView`  
`sprite(whichSprite).camera{(index)}.fieldOfView`

### Description

3D camera property; indicates the angle formed by two rays: one drawn from the camera to the top of the projection plane, and the other drawn from the camera to the bottom of the projection plane.

The images of the models in the 3D world are mapped onto the projection plane, which is positioned in front of the camera like a screen in front of a movie projector. The projection plane is what you see in the 3D sprite. The top and bottom of the projection plane are defined by the `fieldOfView` property. Note, however, that the sprite is not resized as the value of the `fieldOfView` property changes. Instead, the image of the projection plane is scaled to fit the rect of the sprite.

The value of this property is meaningful only when the value of the camera's `projection` property is set to `#perspective`. When the `projection` property is set to `#orthographic`, use the camera's `orthoHeight` property to define the top and bottom of the projection plane.

The default setting for this property is 30.0.

#### Example

This statement sets the `fieldOfView` property of camera 1 to 90:

```
member("3d world").camera[1].fieldOfView = 90
```

#### See also

`orthoHeight`

## fileName (cast property)

### Syntax

`castLib(whichCast).fileName`  
the `fileName` of castLib *whichCast*

### Description

Property; specifies the filename of the specified cast.

- For an external cast, `fileName` gives the cast's full pathname and filename.
- For an internal cast, the `fileName castLib` property depends on which internal cast is specified. For the first internal cast library, the `fileName castLib` property specifies the name of the movie. For remaining internal casts, `fileName` is an empty string.

The `fileName` of `castLib` property accepts URLs as references. However, to use a cast from the Internet and minimize download time, use the `downloadNetThing` or `preloadNetThing` command to download the cast's file to a local disk first and then set `fileName castLib` to the file on the disk.

If a movie sets the filename of an external cast, don't use the Duplicate Cast Members for Faster Loading option in the Projector Options dialog box.

This property can be tested and set for external casts. It can be tested only for internal casts.

**Note:** Director for Java does not support the `downloadNetThing` command.

### Examples

This statement displays the pathname and filename of the Buttons external cast in the Message window:

```
put castLib("Buttons").fileName
```

This statement sets the filename of the Buttons external cast to `Content.cst`:

```
castLib("Buttons").fileName = the moviePath & "Content.cst"
```

The movie then uses the external cast file `Content.cst` as the Buttons cast.

These statements download an external cast from a URL to the Director application folder and then make that file the external cast named `Cast of Thousands`:

```
downloadNetThing("http://www.cbDeMille.com Thousands.cst", the \
    applicationPath & "Thousands.cst")
castLib("Cast of Thousands").fileName = the applicationPath & "Thousands.cst"
```

#### See also

`downloadNetThing`, `preloadNetThing()`



## fileName (cast member property)

### Syntax

`member(whichCastMember).fileName`  
the `fileName` of member *whichCastMember*

### Description

Cast member property; refers to the name of the file assigned to the linked cast member specified by *whichCastMember*. This property is useful for switching the external linked file assigned to a cast member while a movie plays, similar to the way you can switch cast members. When the linked file is in a different folder than the movie, you must include the file's pathname.

You can also make unlinked media linked by setting the filename of those types of members that support linked media.

The `fileName` member property accepts URLs as a reference. However, to use a file from a URL and minimize download time, use the `downloadNetThing` or `preloadNetThing` command to download the file to a local disk first and then set `fileName` member property to the file on the local disk.

The Director player for Java doesn't support the `downloadNetThing` command, so the player can't download files in the background before assigning a new file to a cast member. Changing the `fileName` member property in a movie playing as an applet can make the applet wait for the new file to download.

This property can be tested and set. After the filename is set, Director uses that file the next time the cast member is used.

### Example

This statement links the QuickTime movie "ChairAnimation" to cast member 40:

```
member(40).fileName = "ChairAnimation"
```

These statements download an external file from a URL to the Director application folder and make that file the media for the sound cast member Norma Desmond Speaks:

```
downloadNetThing("http://www.cbDeMille.com/ Talkies.AIF",the \
    applicationPath&"Talkies.AIF")
member("Norma Desmond Speaks").fileName = the applicationPath & "Talkies.AIF"
```

### See also

`downloadNetThing`, `preloadNetThing()`

## fileName (window property)

### Syntax

`window whichWindow.fileName`  
the `fileName` of window *whichWindow*

### Description

Window property; refers to the filename of the movie assigned to the window specified by *whichWindow*. When the linked file is in a different folder than the movie, you must include the file's pathname.

To be able to play the movie in a window, you must set the `fileName` window property to the movie's filename.

The `fileName` of `window` property accepts URLs as a reference. However, to use a movie file from a URL and minimize the download time, use the `downloadNetThing` or `preloadNetThing` command to download the movie file to a local disk first and then set `fileName` window property to the file on the local disk.

This property can be tested and set.

### Examples

This statement assigns the file named Control Panel to the window named Tool Box:

```
window("Tool Box").fileName = "Control Panel"
```

This statement displays the filename of the file assigned to the window named Navigator:

```
put window("Navigator").fileName
```

These statements download a movie file from a URL to the Director application folder and then assign that file to the window named My Close Up:

```
downloadNetThing("http://www.cbDeMille.com/Finale.DIR", the \
    applicationPath&"Finale.DIR")
window("My Close Up").fileName = the applicationPath&"Finale.DIR"
```

### See also

`downloadNetThing`, `preloadNetThing()`

## fill()

### Syntax

```
imageObject.fill(left, top, right, bottom, colorObjectOrParameterList)
imageObject.fill(point(x, y), point(x, y), colorObjectOrParameterList)
imageObject.fill(rect, colorObjectOrParameterList)
```

### Description

This function fills a rectangular region with the color *colorObject* in the given image object. You specify the rectangle in any of the three ways shown. The points specified are relative to the upper-left corner of the given image object. The return value is 1 if there is no error, zero if there is an error.

If you provide a *parameterList* instead of a simple *colorObject*, the rectangle is filled with a shape you specify with these parameters:

Property	Description
<code>#shapeType</code>	A symbol value of <code>#oval</code> , <code>#rect</code> , <code>#roundRect</code> , or <code>#line</code> . The default is <code>#line</code> .
<code>#lineSize</code>	The width of the line to use in drawing the shape.
<code>#color</code>	A color object, which determines the fill color of the shape.
<code>#bgColor</code>	A color object, which determines the color of the shape's border.

For best performance, with 8-bit or lower images the *colorObject* should contain an indexed color value. For 16- or 32-bit images, use an RGB color value.

### Examples

This statement renders the image object in the variable *myImage* completely black:

```
myImage.fill(myImage.rect, rgb(0, 0, 0))
```

The following statement draws a filled oval in the image object TestImage. The oval has a green fill and a 5-pixel-wide red border.

```
TestImage.fill(0, 0, 100, 100, [#shapeType: #oval, #lineSize: 5, #color:  
    rgb(0, 255, 0), \  
    #bgColor: rgb(255, 0, 0)])
```

**See also**

color(), draw()

## fillColor

**Syntax**

member(*whichCastMember*).fillColor

**Description**

Vector shape cast member property; the color of the shape's fill specified as an RGB value.

It's possible to use fillColor when the fillMode property of the shape is set to #solid or #gradient, but not if it is set to #none. If the fillMode is #gradient, fillColor specifies the starting color for the gradient. The ending color is specified with endColor.

This property can be tested and set.

To see an example of fillColor used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

**Example**

This statement sets the fill color of the member Archie to a new RGB value:

```
member("Archie").fillColor = rgb( 24, 15, 153)
```

**See also**

endColor, fillMode

## fillCycles

**Syntax**

member(*whichCastMember*).fillCycles

**Description**

Vector shape cast member property; the number of fill cycles in a gradient vector shape's fill, as specified by an integer value from 1 to 7.

This property is valid only when the fillMode property of the shape is set to #gradient.

This property can be tested and set.

To see an example of fillCycles used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

**Example**

This statement sets the fillCycles of member Archie to 3:

```
member("Archie").fillCycles = 3
```

**See also**

endColor, fillColor, fillMode

## fillDirection

### Syntax

`member(whichCastMember).fillDirection`

### Description

Vector shape cast member property; specifies the amount in degrees to rotate the fill of the shape.

This property is only valid when the `fillMode` property of the shape is set to `#gradient`.

This property can be tested and set.

To see an example of `fillDirection` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

### See also

`fillMode`

## filled

### Syntax

`member(whichCastMember).filled`  
the filled of member *whichCastMember*

### Description

Shape cast member property; indicates whether the specified cast member is filled with a pattern (TRUE) or not (FALSE).

### Example

The following statements make the shape cast member Target Area a filled shape and assign it the pattern numbered 1, which is a solid color:

```
member("Target Area").filled = TRUE  
member("Target Area").pattern = 1
```

### See also

`fillColor`, `fillMode`

## fillMode

### Syntax

`member(whichCastMember).fillMode`

### Description

Vector shape cast member property; indicates the fill method for the shape, using the following possible values:

- `#none`—The shape is transparent
- `#solid`—The shape uses a single fill color
- `#gradient`—The shape uses a gradient between two colors

This property can be tested and set when the shape is closed; open shapes have no fill.

To see an example of `fillMode` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

**Example**

This statement sets the `fillMode` of member Archie to `gradient`:

```
member("Archie").fillMode = #gradient
```

**See also**

`endColor`, `fillColor`

## fillOffset

**Syntax**

```
member(whichCastMember).fillOffset
```

**Description**

Vector shape cast member property; specifies the horizontal and vertical amount in pixels (within the `defaultRect` space) to offset the fill of the shape.

This property is only valid when the `fillMode` property of the shape is set to `#gradient`, but can be both tested and set.

To see an example of `fillOffset` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

**Example**

This statement changes the fill offset of the vector shape cast member `miette` to a horizontal offset of 33 pixels and a vertical offset of 27 pixels:

```
member("miette").fillOffset = point(33, 27)
```

**See also**

`defaultRect`, `fillMode`

## fillScale

**Syntax**

```
member(whichCastMember).fillScale
```

**Description**

Vector shape cast member property; specifies the amount to scale the fill of the shape. This property is referred to as “spread” in the vector shape window.

This property is only valid when the `fillMode` property of the shape is set to `#gradient`, but can be both tested and set.

To see an example of `fillScale` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

**Example**

This statement sets the `fillScale` of member Archie to 33:

```
member("Archie").fillScale = 33.00
```

**See also**

`fillMode`

## findEmpty()

### Syntax

```
findEmpty(member whichCastMember)
```

### Description

Function; for the current cast only, displays the next empty cast member position or the position after the cast member specified by *whichCastMember*.

### Example

This statement finds the first empty cast member on or after cast member 100:

```
put findEmpty(member 100)
```

## findLabel()

### Syntax

```
sprite(whichFlashSprite).findLabel(whichLabelName)  
findLabel(sprite whichFlashSprite, whichLabelName)
```

### Description

Function: this function returns the frame number (within the Flash movie) that is associated with the label name requested.

A 0 is returned if the label doesn't exist, or if that portion of the Flash movie has not yet been streamed in.

## findPos

### Syntax

```
list.findPos(property)  
findPos(list, property)
```

### Description

List command; identifies the position of the property specified by *property* in the property list specified by *list*.

Using `findPos` with linear lists returns a bogus number if the value of *property* is a number and a script error if the value of *property* is a string.

The `findPos` command performs the same function as the `findPosNear` command, except that `findPos` is VOID when the specified property is not in the list.

### Example

This statement identifies the position of the property `c` in the list `Answers`, which consists of `[#a:10, #b:12, #c:15, #d:22]`:

```
Answers.findPos(#c)
```

The result is 3, because `c` is the third property in the list.

### See also

`findPosNear`, `sort`

## findPosNear

### Syntax

```
sortedList.findPosNear(valueOrProperty)  
findPosNear(sortedList, valueOrProperty)
```

### Description

List command; for sorted lists only, identifies the position of the item specified by *valueOrProperty* in the specified sorted list.

The `findPosNear` command works only with sorted lists. Replace *valueOrProperty* with a value for sorted linear lists, and with a property for sorted property lists.

The `findPosNear` command is similar to the `findPos` command, except that when the specified property is not in the list, the `findPosNear` command identifies the position of the value with the most similar alphanumeric name. This command is useful in finding the name that is the closest match in a sorted directory of names.

### Example

This statement identifies the position of a property in the sorted list `Answers`, which consists of `[#Nile:2, #Pharaoh:4, #Raja:0]`:

```
Answers.findPosNear(#Ni)
```

The result is 1, because `Ni` most closely matches `Nile`, the first property in the list.

### See also

`findPos`

## finishIdleLoad

### Syntax

```
finishIdleLoad loadTag
```

### Description

Command; forces completion of loading for all the cast members that have the specified load tag.

### Example

This statement completes the loading of all cast members that have the load tag 20:

```
finishIdleLoad 20
```

### See also

`idleHandlerPeriod`, `idleLoadDone()`, `idleLoadMode`, `idleLoadPeriod`, `idleLoadTag`, `idleReadChunkSize`

## firstIndent

### Syntax

*chunkExpression*.firstIndent

### Description

Text cast member property; contains the number of pixels the first indent in *chunkExpression* is offset from the left margin of the *chunkExpression*.

The value is an integer: less than 0 indicates a hanging indent, 0 is no indentation, and greater than 0 is a normal indentation.

This property can be tested and set.

### Example

This statement sets the indent of the first line of member Desk to 0 pixels:

```
member("Desk").firstIndent = 0
```

### See also

leftIndent, rightIndent

## fixedLineSpace

### Syntax

*chunkExpression*.fixedLineSpace

### Description

Text cast member property; controls the height of each line in the *chunkExpression* portion of the text cast member.

The value itself is an integer, indicating height in absolute pixels of each line.

The default value is 0, which results in natural height of lines.

### Example

This statement sets the height in pixels of each line of member Desk to 24:

```
member("Desk").fixedLineSpace = 24
```

## fixedRate

### Syntax

```
sprite(whichFlashOrGIFSprite).fixedRate  
the fixedRate of sprite whichFlashOrGIFSprite  
member(whichFlashOrGIFMember).fixedRate  
the fixedRate of member whichFlashOrGIFMember
```

### Description

Cast member property and sprite property; controls the frame rate of a Flash movie or animated GIF. The *fixedRate* property can have integer values. The default value is 15.

This property is ignored if the sprite's *playbackMode* property is anything other than *#fixed*.

This property can be tested and set.



### Example

The following handler adjusts the frame rate of a Flash movie sprite. As parameters, the handler accepts a sprite reference, an indication of whether to speed up or slow down the Flash movie, and the amount to adjust the speed.

```
on adjustFixedRate whichSprite, adjustType, howMuch
  case adjustType of
    #faster:
      sprite(whichSprite).fixedRate = sprite(whichSprite).fixedRate + howMuch
    #slower:
      sprite(whichSprite).fixedRate = sprite(whichSprite).fixedRate - howMuch
  end case
end
```

### See also

playBackMode

## fixStageSize

### Syntax

the fixStageSize

### Description

Movie property; determines whether the Stage size remains the same when you load a new movie (TRUE, default), or not (FALSE), regardless of the Stage size saved with that movie, or the setting for the centerStage.

The fixStageSize property cannot change the Stage size for a movie that is currently playing.

This property can be tested and set.

### Examples

The following statement determines whether the fixStageSize property is turned on. If fixStageSize is FALSE, it sends the playhead to a specified frame.

```
if the fixStageSize = FALSE then go to frame "proper size"
```

This statement sets the fixStageSize property to the opposite of its current setting:

```
the fixStageSize = not the fixStageSize
```

### See also

centerStage

## flashRect

### Syntax

```
member(whichVectorOrFlashMember).flashRect  
the flashRect of member whichVectorOrFlashMember
```

### Description

Cast member property; indicates the size of a Flash movie or vector shape cast member as it was originally created. The property values are indicated as a Director rectangle: for example, rect(0,0,32,32).

For linked Flash cast members, the flashRect member property returns a valid value only when the cast member's header has finished loading into memory.

This property can be tested but not set.

### Example

This sprite script resizes a Flash movie sprite so that it is equal to the original size of its Flash movie cast member:

```
on beginSprite me
    sprite(me.spriteNum).rect = sprite(me.spriteNum).member.FlashRect
end
```

### See also

defaultRect, defaultRectMode, state (Flash, SWA)

## flashToStage()

### Syntax

```
sprite(whichFlashSprite).flashToStage(pointInFlashMovie)
flashToStage (sprite whichFlashSprite, pointInFlashMovie)
```

### Description

Function; returns the coordinate on the Director Stage that corresponds to a specified coordinate in a Flash movie sprite. The function accepts both the Flash channel and movie coordinate and returns the Director Stage coordinate as Director point values: for example, point(300,300).

Flash movie coordinates are measured in Flash movie pixels, which are determined by a movie's original size when it was created in Flash. For the purpose of calculating Flash movie coordinates, point(0,0) of a Flash movie is always at its upper left corner. (The cast member's `originPoint` property is used only for rotation and scaling, not to calculate movie coordinates.)

The `flashToStage` and the corresponding `stageToFlash` functions are helpful for determining which Flash movie coordinate is directly over a Director Stage coordinate. For both Flash and Director, point(0,0) is the upper left corner of the Flash Stage or Director Stage. These coordinates may not match on the Director Stage if a Flash sprite is stretched, scaled, or rotated.

### Example

This handler accepts a point value and a sprite reference as a parameter, and it then sets the upper left coordinate of the specified sprite to the specified point within a Flash movie sprite in channel 10:

```
on snapSprite whichFlashPoint, whichSprite
    sprite(whichSprite).loc = sprite(1).FlashToStage(whichFlashPoint)
    updatestage
end
```

### See also

stageToFlash()

## flat

### Syntax

```
member(whichCastmember).shader(whichShader).flat
member(whichCastmember).model(whichModel).shader.flat
member(whichCastmember).model(whichModel).shaderList[[index]].flat
```

### Description

3D #standard shader property; indicates whether the mesh should be rendered with flat shading (TRUE) or Gouraud shading (FALSE).

Flat shading uses one color per face of the mesh. The color used for the face is the color of its first vertex. Flat shading is faster than Gouraud shading.

Gouraud shading assigns a color to each vertex of a face and interpolates the colors across the face in a gradient. Gouraud shading requires more time and calculation, but creates a smoother surface.

The default value for this property is `FALSE`.

#### Example

The following statement sets the `flat` property of the shader named `Wall` to `TRUE`. The mesh of a model that uses this shader will be rendered with one color per face.

```
member("MysteryWorld").shader("Wall").flat = TRUE
```

#### See also

`mesh (property)`, `colors`, `vertices`, `generateNormals()`

## flipH

#### Syntax

```
sprite(whichSpriteNumber).flipH  
the flipH of sprite whichSpriteNumber
```

#### Description

Sprite property; indicates whether a sprite's image has been flipped horizontally on the Stage (`TRUE`) or not (`FALSE`).

The image itself is flipped around its registration point.

This means any rotation or skew remains constant; only the image data itself is flipped.

#### Example

This statement displays the `flipH` of sprite 5:

```
put sprite (5).flipH
```

#### See also

`flipV`, `rotation`, `skew`

## flipV

#### Syntax

```
sprite(whichSpriteNumber).flipV  
the flipV of sprite whichSpriteNumber
```

#### Description

Sprite property; indicates whether a sprite's image has been flipped vertically on the Stage (`TRUE`) or not (`FALSE`).

The image itself is flipped around its registration point.

This means any rotation or skew remains constant; only the image data itself is flipped.

#### Example

This statement displays the `flipV` of sprite 5:

```
sprite (5).flipV = 1
```

#### See also

`flipH`, `rotation`, `skew`

## float()

### Syntax

```
(expression).float  
float (expression)
```

### Description

Function; converts an expression to a floating-point number. The number of digits that follow the decimal point (for display purposes only, calculations are not affected) is set using the `floatPrecision` property.

### Examples

This statement converts the integer 1 to the floating-point number 1:

```
put (1).float  
-- 1.0
```

Math operations can be performed using `float`; if any of the terms is a `float` value, the entire operation is performed with `float`:

```
"the floatPrecision = 1  
put 2 + 2  
-- 4  
put (2).float + 2  
-- 4.0  
the floatPrecision = 4  
put 22/7  
-- 3  
put (22).float / 7  
-- 3.1429"
```

### See also

`floatPrecision`, `ilk()`

## floatP()

### Syntax

```
(expression).floatP  
floatP(expression)
```

### Description

Function; indicates whether the value specified by *expression* is a floating-point number (1 or TRUE) or not (0 or FALSE).

The *P* in `floatP` stands for *predicate*.

### Examples

This statement tests whether 3.0 is a floating-point number. The Message window displays the number 1, indicating that the statement is TRUE.

```
put (3.0).floatP  
-- 1
```

This statement tests whether 3 is a floating-point number. The Message window displays the number 0, indicating that the statement is FALSE.

```
put (3).floatP  
-- 0
```

### See also

`float()`, `ilk()`, `integerP()`, `objectP()`, `stringP()`, `symbolP()`

## floatPrecision

### Syntax

the floatPrecision

### Description

Movie property; rounds off the display of floating-point numbers to the number of decimal places specified. The value of `floatPrecision` must be an integer. The maximum value is 15 significant digits; the default value is 4.

The `floatPrecision` property determines only the number of digits used to display floating-point numbers; it does not change the number of digits used to perform calculations.

- If `floatPrecision` is a number from 1 to 15, floating-point numbers display that number of digits after the decimal point. Trailing zeros remain.
- If `floatPrecision` is zero, floating-point numbers are rounded to the nearest integer. No decimal points appear.
- If `floatPrecision` is a negative number, floating-point numbers are rounded to the absolute value for the number of decimal places. Trailing zeros are dropped.

This property can be tested and set.

### Examples

This statement rounds off the square root of 3.0 to three decimal places:

```
the floatPrecision = 3
x = sqrt(3.0)
put x
-- 1.732
```

This statement rounds off the square root of 3.0 to eight decimal places:

```
the floatPrecision = 8
put x
-- 1.73205081
```

## flushInputEvents

### Syntax

flushInputEvents()

### Description

This command will flush any waiting mouse or keyboard events from the Director message queue. Generally this is useful when Lingo is in a tight repeat loop and the author wants to make sure any mouse clicks or keyboard presses don't get through. This command operates at runtime only and has no effect during authoring.

### Example

This Lingo disables mouse and keyboard events while a repeat loop executes:

```
repeat with i = 1 to 10000
    flushInputEvents()
    sprite(1).loc = sprite(1).loc + point(1, 1)
end repeat

mouseDown, mouseUp, keyDown, keyUp
```

# fog

## Syntax

```
member(whichCastmember).camera(whichCamera).fog.color  
sprite(whichSprite).camera{(index)}.fog.color  
member(whichCastmember).camera(whichCamera).fog.decayMode  
sprite(whichSprite).camera{(index)}.fog.decayMode  
member(whichCastmember).camera(whichCamera).fog.enabled  
sprite(whichSprite).camera{(index)}.fog.enabled  
member(whichCastmember).camera(whichCamera).fog.far  
sprite(whichSprite).camera{(index)}.fog.far  
member(whichCastmember).camera(whichCamera).fog.near  
sprite(whichSprite).camera{(index)}.fog.near
```

## Description

3D camera property; fog introduces a coloring and blurring of models that increases with distance from the camera. The effect is similar to real fog, except that it can be any color.

The “See also” section of this entry contains a complete list of fog properties. See the individual property entries for more information.

## See also

color (fog), decayMode, enabled (fog), far (fog), near (fog)

# font

## Syntax

```
member(whichCastMember).font  
the font of member whichCastMember
```

## Description

Text and field cast member property; determines the font used to display the specified cast member and requires that the cast member contain characters, if only a space. The parameter *whichCastMember* can be either a cast member name or number.

The Director player for Java doesn’t map to other fonts when converting a movie; Java substitutes the default font for any unsupported font. Use only Java’s supported fonts as values for the `font` member property in a movie that plays back as an applet.

Java offers only the following cross-platform fonts:

Java font name	Corresponding Windows font	Corresponding Macintosh font
Helvetica	Arial	Helvetica
TimesRoman	Times New Roman	Times
Courier	Courier-New	Courier
Dialog	MS Sans Serif	Chicago or Charcoal
DialogInput	MS Sans Serif	Geneva
ZapfDingbats	WingDings	Zapf Dingbats
Default	Arial	Helvetica

The `font` member property can be tested and set.

To see an example of `font` used in a completed movie, see the Text movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This statement sets the variable named `oldFont` to the current `font` setting for the field cast member Rokujo Speaks:

```
oldFont = member("Rokujo Speaks").font
```

### See also

`text`, `alignment`, `fontSize`, `fontStyle`, `lineHeight` (cast member property)

## fontSize

### Syntax

`member(whichCastMember).fontSize`  
the `fontSize` of member *whichCastMember*

### Description

Field cast member property; determines the size of the font used to display the specified field cast member and requires that the cast member contain characters, if only a space. The parameter *whichCastMember* can be either a cast member name or number.

This property can be tested and set. When tested, it returns the height of the first line in the field. When set, it affects every line in the field.

To see an example of `fontSize` used in a completed movie, see the Text movie in the Learning/Lingo Examples folder inside the Director application folder.

### Examples

This statement sets the variable named `oldSize` to the current `fontSize` of member setting for the field cast member Rokujo Speaks:

```
oldSize = member("Rokujo Speaks").fontSize
```

This statement sets the third line of the text cast member `myMenu` to 24 points:

```
member("myMenu").fontSize = 12
```

### See also

`text`, `alignment`, `font`, `fontStyle`, `lineHeight` (cast member property)

## fontStyle

### Syntax

`member(whichCastMember).fontStyle`  
the `fontStyle` of member *whichCastMember*  
`member(whichCastMember).char[whichChar].fontStyle`  
the `fontStyle` of char *whichChar*  
`member(whichCastMember).word[whichWord].fontStyle`  
the `fontStyle` of word *whichWord*  
`member(whichCastMember).line[whichLine].fontStyle`  
the `fontStyle` of line *whichLine*

### Description

Cast member property; determines the styles applied to the font used to display the specified field cast member, character, line, word, or other chunk expression and requires that the field cast member contain characters, if only a space.

The value of the property is a string of styles delimited by commas. Lingo uses a font that is a combination of the styles in the string. The available styles are plain, bold, italic, underline, shadow, outline, and extended; on the Macintosh, condensed also is available.

Use the style plain to remove all currently applied styles. The parameter *whichCastMember* can be either a cast member name or number.

For a movie playing back as an applet, plain, bold, and italic are the only valid styles for the `fontStyle` member property. The Director player for Java doesn't support underline, shadow, outline, extended, or condensed font styles.

This property can be tested and set.

To see an example of `fontStyle` used in a completed movie, see the Text movie in the Learning/Lingo Examples folder inside the Director application folder.

### Examples

This statement sets the variable named `oldStyle` to the current `fontStyle` setting for the field cast member Rokujo Speaks:

```
oldStyle = member("Rokujo Speaks").fontStyle
```

This statement sets the `fontStyle` member property for the field cast member Poem to bold italic:

```
member("Poem").fontStyle = [#bold, #italic]
```

This statement sets the `fontStyle` property of the third word of the cast member Son's Names to italic:

```
member("Son's Names").word[3].fontStyle = [#italic]
```

This statement sets the `fontStyle` member property of word 1 through word 4 of text member myNote to bold italic:

```
member("myNote").word[1..4].fontstyle = [#bold, #italic]
```

### See also

`text`, `alignment`, `fontSize`, `font`, `lineHeight` (cast member property)

## foreColor

### Syntax

```
member(castName).foreColor = colorNumber  
set the foreColor of member castName to colorNumber  
sprite whichSprite.foreColor  
the foreColor of sprite whichSprite
```

### Description

Cast member property; sets the foreground color of a field cast member.

For a movie that plays back as an applet, specify colors for the `foreColor` sprite property as the decimal equivalent of the 24-bit hexadecimal values used in an HTML document.



It is not recommended to apply this property to bitmap cast members deeper than 1-bit, as the results are difficult to predict.

It is recommended that the newer `color` property be used instead of the `foreColor` property.

### Examples

The hexadecimal value for pure red, FF0000, is equivalent to 16711680 in decimal numbers. This statement specifies pure red as a cast member's forecolor:

```
member(20).foreColor = 16711680
```

This statement changes the color of the field in cast member 1 to the color in palette entry 250:

```
member(1).foreColor = 250
```

The following statement sets the variable `oldColor` to the foreground color of sprite 5:

```
oldColor = sprite(5).foreColor
```

The following statement makes 36 the number for the foreground color of a random sprite from sprites 11 to 13:

```
sprite(10 + random(3)).foreColor = 36
```

The following statement sets the `foreColor` of word 3 of line 2 of text member `myDescription` to a value of 27:

```
member("myDescription").line[2].word[3].forecolor = 27
```

### See also

`backColor`, `color` (sprite and cast member property)

## forget()

### Syntax

```
timeout("timeoutName").forget()  
forget(timeout("timeoutName"))
```

### Description

This timeout object function removes the given *timeoutObject* from the `timeoutList`, and prevents it from sending further timeout events.

### Example

This statement deletes the timeout object named `AlarmClock` from the `timeoutList`:

```
timeout("AlarmClock").forget()
```

### See also

`timeout()`, `timeoutHandler`, `timeoutList`, `new()`

## forget

### Syntax

```
window(whichWindow).forget()  
forget window whichWindow
```

### Description

Window property; instructs Lingo to close and delete the window specified by *whichWindow* when it's no longer in use and no other variables refer to it.

When a `forget window` command is given, the window and the movie in a window (MIAW) disappear without calling the `on stopMovie`, `on closeWindow`, or `on deactivateWindow` handlers.

If there are many global references to the movie in a window, the window doesn't respond to the `forget` command.

### Example

This statement instructs Lingo to delete the window Control Panel when the movie no longer uses the window:

```
window("Control Panel").forget()
```

### See also

`close window`, `open window`

## frame() (function)

### Syntax

```
the frame
```

### Description

Function; returns the number of the current frame of the movie.

### Example

This statement sends the playhead to the frame before the current frame:

```
go to (the frame - 1)
```

### See also

`go`, `label()`, `marker()`

## frame (sprite property)

### Syntax

```
sprite(whichFlashSprite).frame  
the frame of sprite whichFlashSprite
```

### Description

Sprite property; controls which frame of the current Flash movie is displayed. The default value is 1.

This property can be tested and set.

### Example

The following frame script checks to see if a Flash movie has finished playing (by checking to see if the current frame is equal to the total number of frames in the movie). If the movie has not finished, the playhead continues to loop in the current frame; when the movie finishes, the playhead continues to the next frame. (This script assumes that the movie was designed to stop on its final frame and that it has not been set for looped playback.)

```
on exitFrame
  if sprite(5).frame < sprite(5).member.frameCount then
    go to the frame
  end if
end
```

## frameCount

### Syntax

`member(whichFlashMember).frameCount`  
the frameCount of member *whichFlashMember*

### Description

Flash cast member property; indicates the number of frames in the Flash movie cast member. The frameCount member property can have integer values.

This property can be tested but not set.

### Example

This sprite script displays, in the Message window, the channel number and the number of frames in a Flash movie:

```
property spriteNum

on beginSprite me
  put ""The Flash movie in channel"" && spriteNum && has"" &&
    sprite(spriteNum).member.frameCount && ""frames.""
end
```

## frameLabel

### Syntax

the frameLabel

### Description

Frame property; identifies the label assigned to the current frame. When the current frame has no label, the value of the frameLabel property is 0.

This property can be tested at any time. It can be set during a Score generation session.

### Example

The following statement checks the label of the current frame. In this case, the current frameLabel value is Start:

```
put the frameLabel
-- "Start"
```

### See also

labelList

## framePalette

### Syntax

the framePalette

### Description

Frame property; identifies the cast member number of the palette used in the current frame, which is either the current palette or the palette set in the current frame.

Because the browser controls the palette for the entire Web page, the Director player for Java always uses the browser's palette. For the most reliable color when authoring a movie for playback as a Director player for Java, use the default palette for the authoring system. When you want exact control over colors, use Shockwave instead of Java.

This property can be tested. It can also be set during a Score generation session.

### Examples

The following statement checks the palette used in the current frame. In this case, the palette is cast member 45.

```
put the framePalette
-- 45
```

This statement makes palette cast member 45 the palette for the current frame:

```
the framePalette = 45
```

### See also

puppetPalette

## frameRate

### Syntax

member(*whichCastMember*).frameRate  
the frameRate of member *whichCastMember*

### Description

Cast member property; specifies the playback frame rate for the specified digital video, or Flash movie cast member.

The possible values for the frame rate of a digital video member correspond to the radio buttons for selecting digital video playback options.

- When the `frameRate` member property is between 1 and 255, the digital video movie plays every frame at that frame rate. The `frameRate` member property cannot be greater than 255.
- When the `frameRate` member property is set to -1 or 0, the digital video movie plays every frame at its normal rate. This allows the video to sync to its soundtrack. When the `frameRate` is set to any value other than -1 or 0, the digital video soundtrack will not play.
- When the `frameRate` member property is set to -2, the digital video movie plays every frame as fast as possible.

For Flash movie cast members, the property indicates the frame rate of the movie created in Flash.

This property can be tested but not set.

## Examples

This statement sets the frame rate of the QuickTime digital video cast member Rotating Chair to 30 frames per second:

```
member("Rotating Chair").frameRate = 30
```

This statement instructs the QuickTime digital video cast member Rotating Chair to play every frame as fast as possible:

```
member("Rotating Chair").frameRate = -2
```

The following sprite script checks to see if the sprite's cast member was originally created in Flash with a frame rate of less than 15 frames per second. If the movie's frame rate is slower than 15 frames per second, the script sets the `playBackMode` property for the sprite so it can be set to another rate. The script then sets the sprite's `fixedRate` property to 15 frames per second.

```
property spriteNum
on beginSprite me
  if sprite(spriteNum).member.frameRate < 15 then
    sprite(spriteNum).playBackMode = #fixed
    sprite(spriteNum).fixedRate = 15
  end if
end
```

## See also

`fixedRate`, `movieRate`, `movieTime`, `playBackMode`

# frameReady()

## Syntax

```
frameReady(frameN)
frameReady(frameN, frameZ)
frameReady()
frameReady(sprite whichFlashSprite, frameNumber)
```

## Description

Function; for a Flash movie, determines whether a streaming movie is ready for display. If enough of a sprite has streamed into memory to render the frame (integer for frame number, string for label) specified in the `frameNumber` parameter, this function is `TRUE`; otherwise it is `FALSE`. For a Director movie, this function determines whether all the cast members for *frameN* (the number of the frame) are downloaded from the Internet and available locally.

This function is useful only when streaming a movie, range of frames, cast, or linked cast member. To activate streaming, set the Movie:Playback properties in the Modify menu to Use Media as Available or Show Placeholders.

For Director movies, projectors, and Shockwave movies:

- `frameReady (frameN)`—Determines whether the cast members for *frameN* are downloaded.
- `frameReady (frameN, frameZ)`—Determines whether the cast members for *frameN* through *frameZ* are downloaded.
- `frameReady()`—Determines if cast member used in any frame of the Score are downloaded.

For a demonstration of the `frameReady` function used with a Director movie, see the sample movie “Streaming Shockwave” in Director Help.

This function can be tested but not set.

### Examples

This statement determines whether the cast members for frame 20 are downloaded and ready to be viewed:

```
on exitFrame
  if frameReady(20) then
    -- go to frame 20 if all the required
    --castmembers are locally available
    go to frame 20
  else
    -- resume animating loop while background
    --is streaming
    got to frame 1
  end if
end
```

The following frame script checks to see if frame 25 of a Flash movie sprite in channel 5 can be rendered. If it can't, the script keeps the playhead looping in the current frame of the Director movie. When frame 25 can be rendered, the script starts the movie and lets the playhead proceed to the next frame of the Director movie.

```
on exitFrame
  if the frameReady(sprite 5, 25) = FALSE then
    go to the frame
  else
    play sprite 5
  end if
end
```

### See also

mediaReady

## frameScript

### Syntax

```
the frameScript
```

### Description

Frame property; contains the unique cast member number of the frame script assigned to the current frame.

The `frameScript` property can be tested. During a Score recording session, you can also assign a frame script to the current frame by setting the `frameScript` property.

### Examples

The following statement displays the number of the script assigned to the current frame. In this case, the script number is 25.

```
put the frameScript
-- 25
```

This statement makes the script cast member Button responses the frame script for the current frame:

```
the frameScript = member "Button responses"
```

## frameSound1

### Syntax

```
the frameSound1
```

### Description

Frame property; determines the number of the cast member assigned to the first sound channel in the current frame.

This property can be tested and set. This property can also be set during a Score recording session.

### Example

As part of a Score recording session, this statement assigns the sound cast member Jazz to the first sound channel:

```
the frameSound1 = member("Jazz").number
```

## frameSound2

### Syntax

```
the frameSound2
```

### Description

Frame property; determines the number of the cast member assigned to the second sound channel for the current frame.

This property can be tested and set. This property can also be set during a Score recording session.

### Example

As part of a Score recording session, this statement assigns the sound cast member Jazz to the second sound channel:

```
the frameSound2 = member("Jazz").number
```

## framesToHMS()

### Syntax

```
framesToHMS(frames, tempo, dropFrame, fractionalSeconds)
```

### Description

Function; converts the specified number of frames to their equivalent length in hours, minutes, and seconds. This function is useful for predicting the actual playtime of a movie or controlling a video playback device.

- *frames*—Integer expression that specifies the number of frames.
- *tempo*—Integer expression that specifies the tempo in frames per second.
- *dropFrame*—Compensates for the color NTSC frame rate, which is not exactly 30 frames per second and is meaningful only if FPS is set to 30 frames per second. Normally, this argument is set to FALSE.
- *fractionalSeconds*—Determines whether the residual frames are converted to the nearest hundredth of a second (TRUE) or returned as an integer number of frames (FALSE).

The resulting string uses the form `sHH:MM:SS.FFD`, where:

---

s	A character is used if the time is less than zero, or a space if the time is greater than or equal to zero.
HH	Hours.
MM	Minutes.
SS	Seconds.
FF	Indicates a fraction of a second if <i>fractionalSeconds</i> is TRUE or frames if <i>fractionalSeconds</i> is FALSE.
D	A "d" is used if <i>dropFrame</i> is TRUE, or a space if <i>dropFrame</i> is FALSE.

---

#### Example

The following statement converts a 2710-frame, 30 frame-per-second movie. The *dropFrame* and *fractionalSeconds* arguments are both turned off:

```
put framesToHMS(2710, 30, FALSE, FALSE)
-- " 00:01:30.10 "
```

#### See also

HMStoFrames()

## frameTempo

#### Syntax

the frameTempo

#### Description

Frame property; indicates the tempo assigned to the current frame.

This property can be tested. It can be set during a Score recording session.

#### Example

The following statement checks the tempo used in the current frame. In this case, the tempo is 15 frames per second.

```
put the frameTempo
-- 15
```

#### See also

puppetTempo

## frameTransition

#### Syntax

the frameTransition

#### Description

Frame property; specifies the number of the transition cast member assigned to the current frame.

This property can be set during a Score recording session to specify transitions.



**Example**

When used in a Score recording session, this statement makes the cast member Fog the transition for the frame that Lingo is currently recording:

```
set the frameTransition to member "Fog"
```

## freeBlock()

**Syntax**

```
the freeBlock
```

**Description**

Function; indicates the size of the largest free contiguous block of memory, in bytes. A kilobyte (K) is 1024 bytes. A megabyte (MB) is 1024 kilobytes. Loading a cast member requires a free block at least as large as the cast member.

**Example**

This statement determines whether the largest contiguous free block is smaller than 10K and displays an alert if it is:

```
if (the freeBlock < (10 * 1024)) then alert "Not enough memory!"
```

**See also**

```
freeBytes(), memorySize, ramNeeded(), size
```

## freeBytes()

**Syntax**

```
the freeBytes
```

**Description**

Function; indicates the total number of bytes of free memory, which may not be contiguous. A kilobyte (K) is 1024 bytes. A megabyte (MB) is 1024 kilobytes.

This function differs from `freeBlock` in that it reports all free memory, not just contiguous memory.

On the Macintosh, selecting Use System Temporary Memory in the Director General Preferences or in a projector's Options dialog box tells the `freeBytes` function to return all the free memory that is available to the application. This amount equals the application's allocation shown in its Get Info dialog box and the Largest Unused Block value in the About This Macintosh dialog box.

**Example**

This statement checks whether more than 200K of memory is available and plays a color movie if it is:

```
if (the freeBytes > (200 * 1024)) then play movie "colorMovie"
```

**See also**

```
freeBlock(), memorySize, objectP(), ramNeeded(), size
```

## front

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).front
```

### Description

3D #box model resource property; indicates whether the side of the box intersected by its -Z axis is sealed (TRUE) or open (FALSE).

The default value for this property is TRUE.

### Example

This statement sets the `front` property of the model resource named `Crate` to `FALSE`, meaning the front of this box will be open:

```
member("3D World").modelResource("Crate").front = FALSE
```

### See also

`back`, `bottom` (3D), `top` (3D), `left` (3D), `right` (3D)

## frontWindow

### Syntax

```
the frontWindow
```

### Description

System property; indicates which movie in a window (MIAW) is currently frontmost on the screen. When the Stage is frontmost, `frontWindow` is the Stage. When a media editor or floating palette is frontmost, `frontWindow` returns `VOID`.

This property can be tested but not set.

### Example

This statement determines whether the window "Music" is currently the frontmost window and, if it is, brings the window "Try This" to the front:

```
if the frontWindow = "Music" then window("Try This").moveToFront
```

### See also

`activeWindow`, `on activateWindow`, `on deactivateWindow`, `moveToFront`

## generateNormals()

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).  
generateNormals(style)
```

### Description

3D #mesh model resource command; calculates the normal vectors for each vertex of the mesh.

If the *style* parameter is set to `#flat`, each vertex receives a normal for each face to which it belongs. Furthermore, all three of the vertices of a face will have the same normal. For example, if the vertices of `face[1]` all receive `normal[1]` and the vertices of `face[2]` all receive `normal[2]`, and the two faces share `vertex[8]`, then the normal of `vertex[8]` is `normal[1]` in `face[1]` and `normal[2]` in `face[2]`. Use of the `#flat` parameter results in very clear delineation of the faces of the mesh.

If the *style* parameter is set to `#smooth`, each vertex receives only one normal, regardless of the number of faces to which it belongs, and the three vertices of a face can have different normals. Each vertex normal is the average of the face normals of all of the faces that share the vertex. Use of the `#smooth` parameter results in a more rounded appearance of the faces of the mesh, except at the outer edges of the faces at the silhouette of the mesh, which are still sharp.

A vertex normal is a direction vector which indicates the “forward” direction of a vertex. If the vertex normal points toward the camera, the colors displayed in the area of the mesh controlled by that normal are determined by the shader. If the vertex normal points away from the camera, the area of the mesh controlled by that normal will be non-visible.

After using the `generateNormals()` command, you must use the `build()` command to rebuild the mesh.

#### Example

The following statement calculates vertex normals for the model resource named `FloorMesh`. The *style* parameter is set to `#smooth`, so each vertex in the mesh will receive only one normal.

```
member("Room").modelResource("FloorMesh").generateNormals(#smooth)
```

#### See also

`build()`, `face`, `normalList`, `normals`, `flat`

## getaProp

### Syntax

```
propertyList.propertyName  
getaProp(list, item)  
list[listPosition]  
propertyList [ #propertyName ]  
propertyList [ "propertyName" ]
```

### Description

List command; for linear and property lists, identifies the value associated with the item specified by *item*, *listPosition*, or *propertyName* in the list specified by *list*.

- When the list is a linear list, replace *item* with the number for an item's position in a list as shown by *listPosition*. The result is the value at that position.
- When the list is a property list, replace *item* with a property in the list as in *propertyName*. The result is the value associated with the property.

The `getaProp` command returns `VOID` when the specified value is not in the list.

When used with linear lists, the `getaProp` command has the same function as the `getAt` command.

### Examples

This statement identifies the value associated with the property #joe in the property list ages, which consists of [#john:10, #joe:12, #cheryl:15, #barbara:22]:

```
put getaProp(ages, #joe)
```

The result is 12, because this is the value associated with the property #joe.

The same result can be achieved using bracket access on the same list:

```
put ages[#joe]
```

The result is again 12.

If you want the value at a certain position in the list, you can also use bracket access. To get the third value in the list, associated with the third property, use this syntax:

```
put ages[3]
-- 15
```

**Note:** Unlike the `getAProp` command where `VOID` is returned when a property doesn't exist, a script error will occur if the property doesn't exist when using bracket access.

### See also

`getAt`, `getOne()`, `getProp()`, `setaProp`, `setAt`

## getAt

### Syntax

```
getAt(list, position)
list [position]
```

### Description

List command; identifies the item in the position specified by *position* in the specified list. If the list contains fewer elements than the specified position, a script error occurs.

The `getAt` command works with linear and property lists. This command has the same function as the `getaProp` command for linear lists.

This command is useful for extracting a list from within another list, such as the `deskTopRectList`.

### Examples

This statement causes the Message window to display the third item in the answers list, which consists of [10, 12, 15, 22]:

```
put getAt(answers, 3)
-- 15
```

The same result can be returned using bracket access:

```
put answers[3]
-- 15
```

The following example extracts the first entry in a list containing two entries that specify name, department, and employee number information. Then the second element of the newly extracted list is returned, identifying the department in which the first person in the list is employed. The format of the list is `[["Dennis", "consulting", 510], ["Sherry", "Distribution", 973]]`, and the list is called `employeeInfoList`.

```
firstPerson = getAt(employeeInfoList, 1)
put firstPerson
-- ["Dennis", "consulting", 510]
firstPersonDept = getAt(firstPerson, 2)
put firstPersonDept
-- "consulting"
```

It's also possible to nest `getAt` commands without assigning values to variables in intermediate steps. This format can be more difficult to read and write, but less verbose.

```
firstPersonDept = getAt(getAt(employeeInfoList, 1), 2)
put firstPersonDept
-- "consulting"
```

You can also use the bracket list access:

```
firstPerson = employeeInfoList[1]
put firstPerson
-- ["Dennis", "consulting", 510]
firstPersonDept = firstPerson[2]
put firstPersonDept
-- "consulting"
```

As with `getAt`, brackets can be nested:

```
firstPersonDept = employeeInfoList[1][2]
```

#### See also

`getaProp`, `setaProp`, `setAt`

## on getBehaviorDescription

### Syntax

```
on getBehaviorDescription
    statement(s)
end
```

### Description

System message and event handler; contains Lingo that returns the string that appears in a behavior's description pane in the Behavior Inspector when the behavior is selected.

The description string is optional.

Director sends the `getBehaviorDescription` message to the behaviors attached to a sprite when the Behavior Inspector opens. Place the `on getBehaviorDescription` handler within a behavior.

The handler can contain embedded Return characters for formatting multiple-line descriptions.

### Example

This statement displays "Vertical Multiline textField Scrollbar" in the description pane:

```
on getBehaviorDescription
    return "Vertical Multiline textField Scrollbar"
end
```

### See also

`on getPropertyDescriptionList`, `on getBehaviorTooltip`, `on runPropertyDialog`

## on getBehaviorTooltip

### Syntax

```
on getBehaviorTooltip
    statement(s)
end
```

### Description

System message and event handler; contains Lingo that returns the string that appears in a tooltip for a script in the Library palette.

Director sends the `getBehaviorTooltip` message to the script when the cursor stops over it in the Library palette. Place the `on getBehaviorTooltip` handler within the behavior.

The use of the handler is optional. If no handler is supplied, the cast member name appears in the tooltip.

The handler can contain embedded Return characters for formatting multiple-line descriptions.

### Example

This statement displays “Jigsaw puzzle piece” in the description pane:

```
on getBehaviorTooltip
    return "Jigsaw puzzle piece"
end
```

### See also

`on getPropertyDescriptionList`, `on getBehaviorDescription`, `on runPropertyDialog`

## getBoneID

### Syntax

```
memberReference.modelResource.getBoneID("boneName")
```

### Description

3D model resource property; returns the index number of the bone named *boneName* in the model resource. This property returns 0 if no bone by that name can be found.

### Example

This statement returns an ID number for the bone ShinL:

```
put member("ParkScene").modelResource("LittleKid").getBoneId("ShinL")
-- 40
```

### See also

`bone`

## getError()

### Syntax

```
member(whichSWAmember).getError()
getError(member whichSWAmember)
member(whichFlashmember).getError()
getError(member whichFlashmember)
```

## Description

Function; for Shockwave Audio (SWA) or Flash cast members, indicates whether an error occurred as the cast member streamed into memory and returns a value.

Shockwave Audio cast members have the following possible `getError()` integer values and corresponding `getErrorString()` messages:

<code>getError()</code> value	<code>getErrorString()</code> message
0	OK
1	memory
2	network
3	playback device
99	other

Flash movie cast members have the following possible `getError` values:

- `FALSE`—No error occurred.
- `#memory`—There is not enough memory to load the cast member.
- `#fileNotFound`—The file containing the cast member's assets could not be found.
- `#network`—A network error prevented the cast member from loading.
- `#fileFormat`—The file was found, but it appears to be of the wrong type, or an error occurred while reading the file.
- `#other`—Some other error occurred.

When an error occurs as a cast member streams into memory, Director sets the cast member's state property to -1. Use the `getError` function to determine what type of error occurred.

## Examples

This handler uses `getError` to determine whether an error involving the Shockwave Audio cast member Norma Desmond Speaks occurred and displays the appropriate error string in a field if it did:

```
on exitFrame
  if member("Norma Desmond Speaks").getError() <> 0 then
    member("Display Error Name").text = member("Norma Desmond \
    Speaks").getErrorString()
  end if
end
```

The following handler checks to see whether an error occurred for a Flash cast member named Dali, which was streaming into memory. If an error occurred, and it was a memory error, the script uses the `unloadCast` command to try to free some memory; it then branches the playhead to a frame in the Director movie named Artists, where the Flash movie sprite first appears, so Director can again try to load and play the Flash movie. If something other than an out-of-memory error occurred, the script goes to a frame named Sorry, which explains that the requested Flash movie can't be played.

```
on CheckFlashStatus
    errorCheck = member("Dali").getError()
    if errorCheck <> 0 then
        if errorCheck = #memory then
            member("Dali").clearError()
            unloadCast
            go to frame ("Artists")
        else
            go to frame ("Sorry")
        end if
    end if
end
```

**See also**

`clearError`, `getErrorString()`, `state (Flash, SWA)`

## getError() (XML)

**Syntax**

```
parserObject.getError()
```

**Description**

Function; returns the descriptive error string associated with a given error number (including the line and column number of the XML where the error occurred). When there is no error, this function returns `<VOID>`.

**Example**

These statements check an error after parsing a string containing XML data:

```
errCode = parserObj.parseString(member("XMLtext").text)
errorString = parserObj.getError()
if voidP(errorString) then
    -- Go ahead and use the XML in some way
else
    alert "Sorry, there was an error " & errorString
    -- Exit from the handler
    exit
end if
```

## getErrorString()

**Syntax**

```
member(whichCastMember). getErrorString()
getErrorString(member whichCastMember)
```

**Description**

Function; for Shockwave Audio (SWA) cast members, returns the error message string that corresponds to the error value returned by the `getError()` function.



Possible `getError()` integer values and corresponding `getErrorString()` messages are:

<code>getError()</code> value	<code>getErrorString()</code> message
0	OK
1	memory
2	network
3	playback device
99	other

### Example

This handler uses `getError()` to determine whether an error occurred for Shockwave Audio cast member Norma Desmond Speaks, and if so, uses `getErrorString` to obtain the error message and assign it to a field cast member:

```
on exitFrame
  if member("Norma Desmond Speaks").getError() <> 0 then
    member("Display Error Name").text = member("Norma Desmond \
    Speaks").getErrorString()
  end if
end
```

### See also

`getError()`

## getFlashProperty()

### Syntax

```
sprite(spriteNum).getFlashProperty(targetName, #property)
```

### Description

This function allows Lingo to invoke the Flash action script function `getProperty()` on the given Flash sprite. This Flash action script function is used to get the value of properties of movie clips or levels within a Flash movie. This is similar to testing sprite properties within Director.

The *targetName* is the name of the movie clip or level whose property you want to get within the given Flash sprite.

The *#property* parameter is the name of the property to get. These movie clip properties can be tested: *#posX*, *#posY*, *#scaleX*, *#scaleY*, *#visible*, *#rotate*, *#alpha*, *#name*, *#width*, *#height*, *#target*, *#url*, *#dropTarget*, *#totalFrames*, *#currentFrame*, *#cursor*, and *#lastframeLoaded*.

To get a global property of the Flash sprite, pass an empty string as the *targetName*. These global Flash properties can be tested: *#focusRect* and *#spriteSoundBufferTime*.

See the Flash documentation for descriptions of these properties.

### Example

This statement gets the value of the *#rotate* property of the movie clip Star in the Flash member in sprite 3:

```
sprite(3).setFlashProperty("Star", #rotate)
setFlashProperty()
```

## getFrameLabel()

### Syntax

```
sprite(whichFlashSprite).getFrameLabel(whichFlashFrameNumber)  
getFrameLabel(sprite whichFlashSprite, whichFlashFrameNumber)
```

### Description

Function; returns the frame label within a Flash movie that is associated with the frame number requested. If the label doesn't exist, or that portion of the Flash movie has not yet been streamed in, this function returns an empty string.

### Example

The following handler looks to see if the marker on frame 15 of the Flash movie playing in sprite 1 is called "Lions". If it is, the Director movie navigates to frame "Lions". If it isn't, the Director movie stays in the current frame and the Flash movie continues to play.

```
on exitFrame  
  if sprite(1).getFrameLabel(15) = "Lions" then  
    go "Lions"  
  else  
    go the frame  
  end if  
end
```

## getHardwareInfo()

### Syntax

```
getRendererServices().getHardwareInfo()
```

### Description

3D `rendererServices` method; returns a property list with information about the user's video card. The list contains the following properties:

`#present` is a Boolean value indicating whether the computer has hardware video acceleration.

`#vendor` indicates the name of the manufacturer of the video card.

`#model` indicates the model name of the video card.

`#version` indicates the version of the video driver.

`#maxTextureSize` is a linear list containing the maximum width and height of a texture, in pixels. Textures that exceed this size are downsampled until they do not. To avoid texture sampling artifacts, author textures of various sizes and choose the ones that do not exceed the `#maxTextureSize` value at run time.

`#supportedTextureRenderFormats` is a linear list of texture pixel formats supported by the video card. For details, see `textureRenderFormat`.

`#textureUnits` indicates the number of texture units available to the card.

`#depthBufferRange` is a linear list of bit-depth resolutions to which the `depthBufferDepth` property can be set.

`#colorBufferRange` is a linear list of bit-depth resolutions to which the `colorBufferDepth` property can be set.

### Example

This statement displays a detailed property list of information about the user's hardware:

```
put getRendererServices().getHardwareInfo()
-- [#present: 1, #vendor: "NVIDIA Corporation", #model: \
   "32MB DDR NVIDIA GeForce2 GTS (Dell)", #version: "4.12.01.0532", \
   #maxTextureSize: [2048, 2048], #supportedTextureRenderFormats: \
   [#rgba8888, #rgba8880, #rgba5650, #rgba5551, #rgba5550, \
   #rgba4444], #textureUnits: 2, #depthBufferRange: [16, 24], \
   #colorBufferRange: [16, 32]]
```

### See also

`getRendererServices()`

## getHotSpotRect()

### Syntax

```
sprite(whichQTVRSprite).getHotSpotRect(hotSpotID)
getHotSpotRect(whichQTVRSprite, hotSpotID)
```

### Description

QuickTime VR function; returns an approximate bounding rectangle for the hot spot specified by *hotSpotID*. If the hot spot doesn't exist or isn't visible on the Stage, this function returns `rect(0, 0, 0, 0)`. If the hot spot is partially visible, this function returns the bounding rectangle for the visible portion.

## getLast()

### Syntax

```
list.getLast()
getLast(list)
```

### Description

List function; identifies the last value in a linear or property list specified by *list*.

### Examples

This statement identifies the last item, 22, in the list `Answers`, which consists of [10, 12, 15, 22]:

```
put Answers.getLast()
```

This statement identifies the last item, 850, in the list `Bids`, which consists of [#Gee:750, #Kayne:600, #Ohashi:850]:

```
put Bids.getLast()
```

## getLatestNetID

### Syntax

```
getLatestNetID
```

### Description

This function returns an identifier for the last network operation that started.

The identifier returned by `getLatestNetID` can be used as a parameter in the `netDone`, `netError`, and `netAbort` functions to identify the last network operation.

**Note:** This function is included for backward compatibility. It is recommended that you use the network ID returned from a `net` lingo function rather than `getLatestNetID`. However, if you use `getLatestNetID`, use it immediately after issuing the `netLingo` command.

### Example

This script assigns the network ID of a `getNetText` operation to the field cast member `Result` so results of that operation can be accessed later:

```
on startOperation
    global gNetID
    getNetText("url")
    set gNetID = getLatestNetID()
end
on checkOperation
    global gNetID
    if netDone(gNetID) then
        put netTextResult into member "Result"
    end if
end
```

### See also

`netAbort`, `netDone()`, `netError()`

## getNetText()

### Syntax

```
getNetText(URL {, serverOSSString} {, characterSet})
getNetText(URL, propertyList {, serverOSSString} {, characterSet})
```

### Description

Function; starts the retrieval of text from a file usually on an HTTP or FTP server, or initiates a CGI query.

The first syntax shown starts the text retrieval. You can submit HTTP CGI queries this way and must properly encode them in the URL. The second syntax includes a property list and submits a CGI query, providing the proper URL encoding.

Use the optional parameter *propertyList* to take a property list for CGI queries. The property list is URL encoded and the URL sent is (`urlstring & "?" & encodedproplist`).

Use the optional parameter *serverOSSString* to encode any return characters in *propertylist*. The value defaults to `UNIX` but may be set to `Win` or `Mac` and translates any carriage returns in the *propertylist* argument into those used on the server. For most applications, this setting is unnecessary because line breaks are usually not used in form responses.

The optional parameter *characterSet* applies only if the user is running Director on a shift-JIS (Japanese) system. Possible character set settings are `JIS`, `EUC`, `ASCII`, and `AUTO`. Lingo converts the retrieved data from shift-JIS to the named character set. Using the `AUTO` setting, character set tries to determine what character set the retrieved text is in and translate it to the character set on the local machine. The default setting is `ASCII`.

For a movie that plays back as an applet, the `getNetText` command retrieves text only from the domain that contains the applet. This behavior differs from Shockwave and is necessary due to Java's security model.

Use `netDone` to find out when the `getNetText` operation is complete, and `netError` to find out if the operation was successful. Use `netTextResult` to return the text retrieved by `getNetText`.

The function works with relative URLs.

To see an example of `getNetText()` used in a completed movie, see the Forms and Post movie in the Learning/Lingo Examples folder inside the Director application folder.

### Examples

This script retrieves text from the URL `http://BigServer.com/sample.txt` and updates the field cast member the mouse pointer is on when the mouse button is clicked:

```
property spriteNum
property theNetID

on mouseUp me
    theNetID = getNetText ("http://BigServer.com/sample.txt")
end

on exitFrame me
    if netDone(theNetID) then
        sprite(spriteNum).member.text = netTextResult(theNetID)
    end if
end
```

This example retrieves the results of a CGI query:

```
getNetText("http://www.yourserver.com/cgi-bin/query.cgi?name=Bill")
```

This is the same as the previous example, but it uses a property list to submit a CGI query, and does the URL encoding for you:

```
getNetText("http://www.yourserver.com/cgi-bin/query.cgi", [#name:"Bill"])
```

### See also

`netDone()`, `netError()`, `netTextResult()`

## getNormalized

### Syntax

```
getNormalized(vector)
vector.getNormalized()
```

### Description

3D vector method; copies the vector and divides the x, y, and z components of the copy by the length of the original vector. The resulting vector has a length of 1 world unit.

This method returns the copy and leaves the original vector unchanged. To normalize the original vector, use the `normalize` command.

### Example

The following statement stores the normalized value of the vector `MyVec` in the variable `Norm`. The value of `Norm` is `vector(-0.1199, 0.9928, 0.0000)` and the magnitude of `Norm` is 1.

```
MyVec = vector(-209.9019, 1737.5126, 0.0000)
Norm = MyVec.getNormalized()
put Norm
-- vector( -0.1199, 0.9928, 0.0000 )
put Norm.magnitude
-- 1.0000
```

### See also

`normalize`

## getNthFileNameInFolder()

### Syntax

```
getNthFileNameInFolder(folderPath, fileNumber)
```

### Description

Function; returns a filename from the directory folder based on the specified path and number within the folder. To be found by the `getNthFileNameInFolder` function, Director movies must be set to visible in the folder structure. (On the Macintosh, other types of files are found whether they are visible or invisible.) If this function returns an empty string, you have specified a number greater than the number of files in the folder.

The `getNthFileNameInFolder` function doesn't work with URLs.

To specify other folder names, use the `@ pathname` operator or the full path defined in the format for the specific platform on which the movie is running. For example:

- In Windows, use a directory path such as `C:/Director/Movies`.
- On the Macintosh, use a pathname such as `HardDisk:Director:Movies`. To look for files on the Macintosh desktop, use the path `HardDisk:Desktop Folder`
- This function is not available in Shockwave.

### Example

The following handler returns a list of filenames in the folder on the current path. To call the function, use parentheses, as in `put currentFolder()`.

```
on currentFolder
  fileList = [ ]
  repeat with i = 1 to 100
    n = getNthFileNameInFolder(the moviePath, i)
    if n = EMPTY then exit repeat
    fileList.append(n)
  end repeat
  return fileList
end currentFolder
```

### See also

`@ (pathname)`

## getOne()

### Syntax

```
list.getOne(value)
getOne(list, value)
```

### Description

List function; identifies the position (linear list) or property (property list) associated with the value specified by *value* in the list specified by *list*.

For values contained in the list more than once, only the first occurrence is displayed. The `getOne` command returns the result 0 when the specified value is not in the list.

When used with linear lists, the `getOne` command performs the same functions as the `getPos` command.

### Examples

This statement identifies the position of the value 12 in the linear list Answers, which consists of [10, 12, 15, 22]:

```
put Answers.getOne(12)
```

The result is 2, because 12 is the second value in the list.

This statement identifies the property associated with the value 12 in the property list Answers, which consists of [#a:10, #b:12, #c:15, #d:22]:

```
put Answers.getOne(12)
```

The result is #b, which is the property associated with the value 12.

### See also

`getPos()`

## getPixel()

### Syntax

```
imageObject.getPixel(x, y {, #integer})  
imageObject.getPixel(point(x, y) {, #integer})
```

### Description

This function returns the color value of the pixel at the specified point in the given image object. This value is normally returned as an indexed or RGB color object, depending on the bit depth of the image.

If you include the optional parameter value #integer, however, it's returned as a raw number. If you're setting a lot pixels to the color of another pixel, it's faster to set them as raw numbers. Raw integer color values are also useful because they contain alpha layer information as well as color when the image is 32-bit. The alpha channel information can be extracted from the raw integer by dividing the integer by 2<sup>8+8+8</sup>.

`GetPixel()` returns 0 if the given pixel is outside the specified image object.

### Examples

These statements get the color of the pixel at point (90, 20) in member Happy and set sprite 2 to that color:

```
myColor=member("Happy").image.getPixel(90, 20)  
sprite(2).color=myColor
```

This statement sets the variable alpha to the alpha channel value of the point (25, 33) in the 32-bit image object myImage:

```
alpha = myImage.getPixel(25, 33, #integer) / power(2, 8+8+8)
```

### See also

`depth`, `color()`, `setPixel()`, `power()`

## getPlaylist()

### Syntax

```
sound(channelNum).getPlaylist()  
getPlaylist(sound(channelNum))
```

### Description

This function returns a copy of the list of queued sounds for *soundObject*. This list does not include the currently playing sound.

The list of queued sounds may not be edited directly. You must use `setPlayList()`.

The playlist is a linear list of property lists. Each property list corresponds to one queued sound cast member. Each queued sound may specify these properties:

Property	Description
#member	The sound cast member to play. This property will always be present; all others are optional.
#startTime	The time within the sound at which playback begins, in milliseconds. See <code>startTime</code> .
#endTime	The time within the sound at which playback ends, in milliseconds. See <code>endTime</code> .
#loopCount	The number of times to play a portion of the sound. See <code>loopCount</code> .
#loopStartTime	The time within the sound at which a loop begins, in milliseconds. See <code>loopStartTime</code> .
#loopEndTime	The time within the sound at which a loop ends, in milliseconds. See <code>loopEndTime</code> .
#preloadTime	The amount of the sound to buffer before playback, in milliseconds. See <code>preloadTime</code> .

### Example

The following handler queues two sounds in sound channel 2, starts playing them, and then displays the `playList` in the message window. The playlist includes only the second sound queued, because the first sound is already playing.

```
on playMusic  
    sound(2).queue([#member:member("chimes")])  
    sound(2).queue([#member:member("introMusic"), #startTime:3000,\  
        #endTime:10000, #loopCount:5,#loopStartTime:8000, #loopEndTime:8900])  
    sound(2).play()  
    put sound(2).getPlaylist()  
end  
-- [[#member: (member 12 of castLib 2), #startTime: 3000, #endTime: 10000,  
    #loopCount: 5, #loopStartTime: 8000, #loopEndTime: 8900]]
```

### See also

`endTime`, `loopCount`, `loopEndTime`, `loopStartTime`, `member` (keyword), `preloadTime`, `queue()`, `setPlaylist()`, `startTime`



## getPos()

### Syntax

```
list.getPos(value)  
getPos(list, value)
```

### Description

List function; identifies the position of the value specified by *value* in the list specified by *list*. When the specified value is not in the list, the `getPos` command returns the value 0.

For values contained in the list more than once, only the first occurrence is displayed. This command performs the same function as the `getOne` command when used for linear lists.

### Example

This statement identifies the position of the value 12 in the list `Answers`, which consists of [#a:10, #b:12, #c:15, #d:22]:

```
put Answers.getPos(12)
```

The result is 2, because 12 is the second value in the list.

### See also

`getOne()`

## getPref()

### Syntax

```
getPref(prefFileName)
```

### Description

Function; retrieves the content of the specified file.

When you use this function, replace *prefFileName* with the name of a file created by the `setPref` function. If no such file exists, `getPref` returns `VOID`.

The filename used for *prefFileName* must be a valid filename only, not a full path; Director supplies the path. The path to the file is handled by Director. The only valid file extensions for *prefFileName* are `.txt` and `.htm`; any other extension is rejected.

Do not use this command to access read-only or locked media.

**Note:** In a browser, data written by `setPref` is not private. Any Shockwave movie can read this information and upload it to a server. Confidential information should not be stored using `setPref`.

To see an example of `getPref()` used in a completed movie, see the `Read and Write Text` movie in the `Learning/Lingo Examples` folder inside the Director application folder.

### Example

This handler retrieves the content of the file `Test` and then assigns the file's text to the field `Total Score`:

```
on mouseUp  
    theText = getPref("Test")  
    member("Total Score").text = theText  
end
```

### See also

`setPref`

## getProp()

### Syntax

```
getProp(list, property)  
list.property
```

### Description

Property list function; identifies the value associated with the property specified by *property* in the property list specified by *list*.

Almost identical to the `getaProp` command, the `getProp` command displays an error message if the specified property is not in the list or if you specify a linear list.

### Example

This statement identifies the value associated with the property `#c` in the property list `Answers`, which consists of `[#a:10, #b:12, #c:15, #d:22]`:

```
getProp(Answers, #c)
```

The result is 15, because 15 is the value associated with `#c`.

### See also

`getOne()`

## getPropAt()

### Syntax

```
list.getPropAt(index)  
getPropAt(list, index)
```

### Description

Property list function; for property lists only, identifies the property name associated with the position specified by *index* in the property list specified by *list*. If the specified item isn't in the list, or if you use `getPropAt()` with a linear list, a script error occurs.

### Example

This statement displays the second property in the given list:

```
put Answers.getPropAt(2)  
-- #b
```

The result is 20, which is the value associated with `#b`.

## on getPropertyDescriptionList

### Syntax

```
on getPropertyDescriptionList  
    statement(s)  
end
```

### Description

System message and event handler; contains Lingo that generates a list of definitions and labels for the parameters that appear in a behavior's Parameters dialog box.

Place the `on getPropertyDescriptionList` handler within a behavior script. Behaviors that don't contain an `on getPropertyDescriptionList` handler don't appear in the Parameters dialog box and can't be edited from the Director interface.

The `on getPropertyDescriptionList` message is sent when any action that causes the Behavior Inspector to open occurs: either when the user drags a behavior to the Score or the user double-clicks a behavior in the Behavior Inspector.

The `#default`, `#format`, and `#comment` settings are mandatory for each parameter. The following are possible values for these settings:

---

<code>#default</code>	The parameter's initial setting.
<code>#format</code>	<code>#integer</code> <code>#float</code> <code>#string</code> <code>#symbol</code> <code>#member</code> <code>#bitmap</code> <code>#filmloop</code> <code>#field</code> <code>#palette</code> <code>#picture</code> <code>#sound</code> <code>#button</code> <code>#shape</code> <code>#movie</code> <code>#digitalvideo</code> <code>#script</code> <code>#richtext</code> <code>#ole</code> <code>#transition</code> <code>#extra</code> <code>#frame</code> <code>#marker</code> <code>#ink</code> <code>#boolean</code>
<code>#comment</code>	A descriptive string that appears to the left of the parameter's editable field in the Parameters dialog box.
<code>#range</code>	A range of possible values that can be assigned to a property. The range is specified as a linear list with several values or as a minimum and maximum in the form of a property list: <code>[#min: minValue, #max: maxValue]</code> .

---

### Example

The following handler defines a behavior's parameters that appear in the Parameters dialog box. Each statement that begins with `addProp` adds a parameter to the list named `description`. Each element added to the list defines a property and the property's `#default`, `#format`, and `#comment` values:

```
on getPropertyDescriptionList
    description = [:]
    addProp description,#dynamic, [#default:1, #format:#boolean,
    #comment:"Dynamic"]
    addProp description,#fieldNum, [#default:1, #format:#integer, \
    #comment:"Scroll which sprite:"]
    addProp description, #extentSprite,[#default:1,#format:#integer, \
    #comment: "Extend Sprite:"]
    addProp description,#proportional,[#default:1,#format:#boolean, \
    #comment: "Proportional:"]
    return description
end
```

### See also

`addProp`, `on getBehaviorDescription`, `on runPropertyDialog`

## getRendererServices()

### Syntax

```
getRendererServices()
getRendererServices().whichGetRendererServicesProperty
```

### Description

3D command; returns the `rendererServices` object. This object contains hardware information and properties that affect all 3D sprites and cast members.

The `rendererServices` object has the following properties:

- `renderer` indicates the software rasterizer used to render all 3D sprites.
- `rendererDeviceList` returns a list of software rasterizers available on the user's system. Possible values include `#openGL`, `#directX5_2`, `#directX7_0`, and `#software`. The value of `renderer` must be one of these. This property can be tested but not set.

- `textureRenderFormat` indicates the pixel format used by the renderer. Possible values include `#rgba8888`, `#rgba8880`, `#rgba5650`, `#rgba5550`, `#rgba5551`, and `#rgba4444`. The four digits in each symbol indicate how many bits are used for each red, green, blue, and alpha component.
- `depthBufferDepth` indicates the bit depth of the hardware output buffer.
- `colorBufferDepth` indicates the bit depth of the color buffer. This property can be tested but not set.
- `modifiers` is a linear list of modifiers available for use by models in 3D cast members. Possible values include `#collision`, `#bonesPlayer`, `#keyframePlayer`, `#toon`, `#lod`, `#meshDeform`, `#sds`, `#inker`, and third-party Xtra-based modifiers. This property can be tested but not set.
- `primitives` is a linear list of primitive types available for use in the creation of new model resources. Possible values include `#sphere`, `#box`, `#cylinder`, `#plane`, `#particle`, and third-party Xtra-based primitive types. This property can be tested but not set.

**Note:** For more detailed information about these properties, see the individual property entries.

#### See also

`renderer`, `preferred3DRenderer`, `active3dRenderer`, `rendererDeviceList`

## getStreamStatus()

### Syntax

```
getStreamStatus(netID)
getStreamStatus(URLString)
```

### Description

Function; returns a property list matching the format used for the globally available `tellStreamStatus` function that can be used with callbacks to sprites or objects. The list contains the following strings:

---

<code>#URL</code>	String containing the URL location used to start the network operation.
<code>#state</code>	String consisting of Connecting, Started, InProgress, Complete, "Error", or "NoInformation" (this last string is for the condition when either the net ID is so old that the status information has been dropped or the URL specified in <code>URLString</code> was not found in the cache).
<code>#bytesSoFar</code>	Number of bytes retrieved from the network so far.
<code>#bytesTotal</code>	Total number of bytes in the stream, if known. The value may be 0 if the HTTP server does not include the content length in the MIME header.
<code>#error</code>	String containing "" (EMPTY) if the download is not complete, OK if it completed successfully, or an error code if the download ended with an error.

---

For example, you can start a network operation with `getNetText()` and track its progress with `getStreamStatus()`.

### Example

This statement displays in the message window the current status of a download begun with `getNetText()` and the resulting net ID placed in the variable `netID`:

```
put getStreamStatus(netID)
-- [#URL: "www.macromedia.com", #state: "InProgress", #bytesSoFar: 250, \
   #bytesTotal: 50000, #error: EMPTY]
```

### See also

`on streamStatus`, `tellStreamStatus()`

## getVariable()

### Syntax

```
sprite(flashSpriteNum).getVariable("variableName" {, returnValueOrReference})  
getVariable(sprite flashSpriteNum, "variableName" {, returnValueOrReference})
```

### Description

Function; returns the current value of the given variable from the specified Flash sprite. Flash variables were introduced in Flash version 4.

This function can be used in two ways.

Setting the optional *returnValueOrReference* parameter to TRUE (the default) returns the current value of the variable as a string. Setting the *returnValueOrReference* parameter to FALSE returns the current literal value of the Flash variable.

If the value of the Flash variable is an object reference, you must set the *returnValueOrReference* parameter to FALSE in order for the returned value to have meaning as an object reference. If it is returned as a string, the string will not be a valid object reference.

### Examples

This statement sets the variable `tValue` to the string value of the Flash variable named `gOtherVar` in the Flash movie in sprite 3:

```
tValue = sprite(3).getVariable("gOtherVar",TRUE)  
  
put tValue  
-- "5"
```

This statement sets the variable `tObject` to refer to the same object that the variable named `gVar` refers to in the Flash movie in sprite 3:

```
tObject = sprite(3).getVariable("gVar",FALSE)
```

This statement returns the value of the variable `currentURL` from the Flash cast member in sprite 3 and displays it in the Message window:

```
put getVariable(sprite 3, "currentURL")  
-- "http://www.macromedia.com/software/flash/"
```

### See also

`setVariable()`

## getWorldTransform()

### Syntax

```
member(whichCastmember).node(whichNode).getWorldTransform()  
member(whichCastmember).node(whichNode).getWorldTransform().\  
    position  
member(whichCastmember).node(whichNode).getWorldTransform().\  
    rotation  
member(whichCastmember).node(whichNode).getWorldTransform().scale
```

### Description

3D command; returns the world-relative transform of the model, group, camera, or light represented by node.

The `transform` property of a node is calculated relative to the transform of the node's parent, and is therefore parent-relative. The `getWorldTransform()` command calculates the node's transform relative to the origin of the 3D world, and is therefore world-relative.

Use `member(whichCastmember).node(whichNode).getWorldTransform().position` to find the position property of the node's world-relative transform. You can also use `worldPosition` as a shortcut for `getWorldTransform().position`.

Use `member(whichCastmember).node(whichNode).getWorldTransform().rotation` to find the rotation property of the node's world-relative transform.

Use `member(whichCastmember).node(whichNode).getWorldTransform().scale` to find the scale property of the node's world-relative transform.

These properties can be tested but not set.

### Example

This statement shows the world-relative transform of the model named Box, followed by its position and rotation properties:

```
put member("3d world").model("Box").getworldTransform()
-- transform(1.000000,0.000000,0.000000,0.000000, \
  0.000000,1.000000,0.000000,0.000000, \
  0.000000,0.000000,1.000000,0.000000, - \
  94.144844,119.012825,0.000000,1.000000)
put member("3d world").model("Box"). getworldTransform().position
-- vector(-94.1448, 119.0128, 0.0000)
put member("3d world").model("Box"). getworldTransform().rotation
--vector(0.0000, 0.0000, 0.0000)
```

### See also

`worldPosition`, `transform` (property)

## global

### Syntax

```
global variable1 {, variable2} {, variable3}...
```

### Description

**Keyword;** defines a variable as a global variable so that other handlers or movies can share it.

Every handler that examines or changes the content of a global variable must use the `global` keyword to identify the variable as global. Otherwise, the handler treats the variable as a local variable, even if it is declared to be global in another handler.

**Note:** To ensure that global variables are available throughout a movie, declare and initialize them in the `prepareMovie` handler. Then, if you leave and return to the movie from another movie, your global variables will be reset to the initial values unless you first check to see that they aren't already set.

A global variable can be declared in any handler or script. Its value can be used by any other handlers or scripts that also declare the variable as global. If the script changes the variable's value, the new value is available to every other handler that treats the variable as global.

A global variable is available in any script or movie, regardless of where it is first declared; it is not automatically cleared when you navigate to another frame, movie, or window.

Any variables manipulated in the Message window are automatically global, even though they are not explicitly declared as such.

Shockwave movies playing on the Internet cannot access global variables within other movies, even movies playing on the same HTML page. The only way movies can share global variables is if an embedded movie navigates to another movie and replaces itself through either `goToNetMovie` or `go movie`.

### Example

The following example sets the global variable `StartingPoint` to an initial value of 1 if it doesn't already contain a value. This allows navigation to and from the movie without loss of stored data.

```
global gStartingPoint
on prepareMovie
  if voidP(gStartingPoint) then gStartingPoint = 1
end
```

### See also

`showGlobals`, `property`, `gotoNetMovie`

## globals

### Syntax

the `globals`

### Description

System property; this property contains a special property list of all current global variables with a value other than `VOID`. Each global variable is a property in the list, with the associated paired value.

You can use the following list operations on `globals`:

- `count()`—Returns the number of entries in the list.
- `getPropAt(n)`—Returns the name of the *n*th entry.
- `getProp(x)`—Returns the value of an entry with the specified name.
- `getAProp(x)`—Returns the value of an entry with the specified name.

**Note:** The `globals` property automatically contains the property `#version`, which is the version of Director running. This means there will always be at least one entry in the list, even if no global variables have been declared yet.

This property differs from `showGlobals` in that the `globals` can be used in contexts other than the Message window. To display the `globals` in the Message window, use `showGlobals`.

### See also

`showGlobals`, `clearGlobals`

## glossMap

### Syntax

```
member(whichCastmember).shader(whichShader).glossMap
member(whichCastmember).model(whichModel).shader.glossMap
member(whichCastmember).model(whichModel).shaderList[[index]].\
  glossMap
```

### Description

3D `#standard` shader property; specifies the texture to use for gloss mapping.

When you set this property, the following properties are automatically set:

- The fourth texture layer of the shader is set to the texture you specified.
- The value of `textureModelList[4]` is set to `#none`.
- The value of `blendFunctionList[4]` is set to `#multiply`.

#### Example

This statement sets the texture named `Oval` as the `glossMap` value for the shader used by the model named `GlassBox`:

```
member("3DPlanet").model("GlassBox").shader.glossMap = \  
    member("3DPlanet").texture("Oval")
```

#### See also

`blendFunctionList`, `textureModelList`, `region`, `specularLightMap`, `diffuseLightMap`

## gravity

#### Syntax

```
member(whichCastmember).modelResource(whichModelResource).gravity
```

#### Description

3D particle model resource property; when used with a model resource whose type is `#particle`, allows you to get or set the `gravity` property of the resource as a vector.

This property defines the gravity force applied to all particles in each simulation step.

The default value for this property is `vector(0,0,0)`.

#### Example

In this example, `ThermoSystem` is a model resource of the type `#particle`. This statement sets the gravity property of `ThermoSystem` to the vector `(0, -1, 0)`, which pulls the particles of `thermoSystem` gently down the y axis.

```
member("Fires").modelResource("ThermoSystem").gravity = \  
    vector(0, -1, 0)
```

#### See also

`drag`, `wind`

## go

#### Syntax

```
go {to} {frame} whichFrame  
go {to} movie whichMovie  
go {to} {frame} whichFrame of movie whichMovie
```

#### Description

Command; causes the playhead to branch to the frame specified by *whichFrame* in the movie specified by *whichMovie*. The expression *whichFrame* can be a marker label or an integer frame number. The expression *whichMovie* must specify a movie file. (If the movie is in another folder, *whichMovie* must specify the path.)



The phrase `go loop` tells the playhead to loop to the previous marker and is a convenient means of keeping the playhead in the same section of the movie while Lingo remains active and avoids the use of `go to the frame` in a frame that has a transition which would slow the movie and overwhelm the processor.

It is best to refer to marker labels instead of frame numbers; editing a movie can cause frame numbers to change. Using marker labels also makes it easier to read scripts.

The `go to movie` command loads frame 1 of the movie. If the command is called from within a handler, the handler in which it is placed continues executing. To suspend the handler while playing the movie, use the `play` command, which may be followed by a subsequent `play done` to return.

When you specify a movie to play, specify its path if the movie is in a different folder, but to prevent a potential load failure, don't include the movie's `.dir`, `.dxr` or `.dcr` file extension.

To more efficiently go to a movie at a URL, use the `downloadNetThing` command to download the movie file to a local disk first and then use the `go to movie` command to go to that movie on the local disk.

The following are reset when a movie is loaded: `beepOn` and `constraint` properties; `keyDownScript`, `mouseDownScript`, and `mouseUpScript`; `cursor` and `immediate sprite` properties; `cursor` and `puppetSprite` commands; and custom menus. However, the `timeoutScript` is not reset when loading a movie.

### Examples

This statement sends the playhead to the marker named `start`:

```
go to "start"
```

This statement sends the playhead to the marker named `Memory` in the movie named `Noh Tale to Tell`:

```
go frame("Memory") of movie("Noh Tale to Tell")
```

The following handler tells the movie to loop in the current frame. This handler is useful for making the movie wait in a frame while it plays so the movie can respond to events.

```
on exitFrame
    go the frame
end
```

### See also

`downloadNetThing`, `gotoNetMovie`, `label()`, `marker()`, `pathName` (movie property), `play`, `play done`

## go loop

### Syntax

```
go loop
```

### Description

Command; sends the playhead to the previous marker in the movie, either one marker back from the current frame if the current frame does not have a marker, or to the current frame if the current frame has a marker.

**Note:** This command is equivalent to `marker(0)` in versions of Director prior to Director 7.

If no markers are to the left of the playhead, the playhead branches to:

- The next marker to the right if the current frame does not have a marker.
- The current frame if the current frame has a marker.
- Frame 1 if the movie contains no markers.

The `go loop` command is equivalent to the statement `go to the marker(0)` used in earlier versions of Lingo.

#### **Example**

This statement causes the movie to loop between the current frame and the previous marker:

```
go loop
```

#### **See also**

`go`, `go next`, `go previous`

## **go next**

#### **Syntax**

```
go next
```

#### **Description**

Command; sends the playhead to the next marker in the movie. If no markers are to the right of the playhead, the playhead goes to the last marker in the movie or to frame 1 if there are no markers in the movie.

The `go next` command is equivalent to the statement `go marker(1)` that was used in earlier versions of Lingo.

#### **Example**

This statement sends the playhead to the next marker in the movie:

```
go next
```

#### **See also**

`go`, `go loop`, `go previous`

## **go previous**

#### **Syntax**

```
go previous
```

#### **Description**

Command; sends the playhead to the previous marker in the movie. This marker is two markers back from the current frame if the current frame does not have a marker or one marker back from the current frame if the current frame has a marker.

**Note:** This command is equivalent to `marker(-1)` in previous versions of Director.

If no markers are to the left of the playhead, the playhead branches to one of the following:

- The next marker to the right if the current frame does not have a marker
- The current frame if the current frame has a marker
- Frame 1 if the movie contains no markers

**Example**

This statement sends the playhead to the previous marker in the movie:

```
go previous
```

**See also**

```
go, go next, go loop
```

## goToFrame

**Syntax**

```
sprite(whichFlashSprite).goToFrame(frameNumber)
goToFrame(sprite whichFlashSprite, frameNumber)
sprite(whichFlashSprite).goToFrame(labelNameString)
goToFrame(sprite whichFlashSprite, labelNameString)
```

**Description**

Command; plays a Flash movie sprite beginning at the frame identified by the *frameNumber* parameter. You can identify the frame by either an integer indicating a frame number or by a string indicating a label name. Using the `goToFrame` command has the same effect as setting a Flash movie sprite's `frame` property.

**Example**

The following handler branches to different points within a Flash movie in channel 5. It accepts a parameter that indicates which frame to go to.

```
on Navigate whereTo
    sprite(5).goToFrame(whereTo)
end
```

## gotoNetMovie

**Syntax**

```
gotoNetMovie URL
gotoNetMovie (URL)
```

**Description**

Command; retrieves and plays a new Shockwave movie from an HTTP or FTP server. The current movie continues to run until the new movie is available.

Only URLs are supported as valid parameters. The URL can specify either a filename or a marker within a movie. Relative URLs work if the movie is on an Internet server, but you must include the extension with the filename.

When performing testing on a local disk or network, media must be located in a directory named `dswmedia`.

If a `gotoNetMovie` operation is in progress and you issue a second `gotoNetMovie` command before the first is finished, the second command cancels the first.

## Examples

In this statement, the URL indicates a Director filename:

```
gotoNetMovie "http://www.yourserver.com/movies/movie1.dcr"
```

In this statement, the URL indicates a marker within a filename:

```
gotoNetMovie "http://www.yourserver.com/movies/buttons.dcr#Contents"
```

In the following statement, `gotoNetMovie` is used as a function. The function returns the network ID for the operation.

```
myNetID = gotoNetMovie ("http://www.yourserver.com/movies/  
  buttons.dcr#Contents")
```

## gotoNetPage

### Syntax

```
gotoNetPage "URL", {"targetName"}
```

### Description

Command; opens a Shockwave movie or another MIME file in the browser.

Only URLs are supported as valid parameters. Relative URLs work if the movie is on an HTTP or FTP server.

The *targetName* argument is an optional HTML parameter that identifies the frame or window in which the page is loaded.

- If *targetName* is a window or frame in the browser, `gotoNetPage` replaces the contents of that window or frame.
- If *targetName* isn't a frame or window that is currently open, `goToNetPage` opens a new window. Using the string `"_new"` always opens a new window.
- If *targetName* is not included, `gotoNetPage` replaces the current page, wherever it is located.

In the authoring environment, the `gotoNetPage` command launches the preferred browser if it is enabled. In projectors, this command tries to launch the preferred browser set with the Network Preferences dialog box or `browserName` command. If neither has been used to set the preferred browser, the `goToNetPage` command attempts to find a browser on the computer.

### Examples

The following script loads the file `Newpage.html` into the frame or window named `frwin`. If a window or frame in the current window called `frwin` exists, that window or frame is used. If the window `frwin` doesn't exist, a new window named `frwin` is created.

```
on keyDown  
  gotoNetPage "Newpage.html", "frwin"  
end
```

This handler opens a new window regardless of what window the browser currently has open:

```
on mouseUp  
  gotoNetPage "Todays_News.html", "_new"  
end
```

### See also

`browserName()`, `netDone()`

## gradientType

### Syntax

`member(whichCastMember).gradientType`

### Description

Vector shape cast member property; specifies the actual gradient used in the cast member's fill.

Possible values are `#linear` or `#radial`. The `gradientType` is only valid when the `fillMode` is set to `#gradient`.

This property can be tested and set.

### Example

This handler toggles between linear and radial gradients in cast member "backdrop":

```
on mouseUp me
  if member("backdrop").gradientType = #radial then
    member("backdrop").gradientType = #linear
  else
    member("backdrop").gradientType = #radial
  end if
end
```

### See also

`fillMode`

## group

### Syntax

`member(whichCastmember).group(whichGroup)`  
`member(whichCastmember).group[index]`

### Description

3D element; a node in the 3D world that has a name, transform, parent, and children, but no other properties.

Every 3D cast member has a default group named `World` that cannot be deleted. The parent hierarchy of all models, lights, cameras, and groups that exist in the 3D world terminates in `group("world")`.

### Examples

This statement shows that the fourth group of the cast member `newAlien` is the group `Direct01`:

```
put member("newAlien").group[4]
-- group("Direct01")
```

### See also

`newGroup`, `deleteGroup`, `child`, `parent`

## halt

### Syntax

`halt`

### Description

Command; exits the current handler and any handler that called it and stops the movie during authoring or quits the projector during run time from a projector.

### Example

This statement checks whether the amount of free memory is less than 50K and, if it is, exits all handlers that called it and then stops the movie:

```
if the freeBytes < 50*1024 then halt
```

### See also

`abort`, `exit`, `pass`, `quit`

## handler()

### Syntax

```
scriptObject.handler(#handlerSymbol)
```

### Description

This function returns TRUE if the given *scriptObject* contains a handler whose name is *#handlerSymbol*, and FALSE if it does not. The script object must be a parent script, a child object, or a behavior.

### Example

This Lingo code invokes a handler on an object only if that handler exists:

```
if spiderObject.handler(#pounce) = TRUE then
    spiderObject.pounce()
end if
```

### See also

`handlers()`, `new()`, `rawNew()`, `script`

## handlers()

### Syntax

```
scriptObject.handlers()
```

### Description

This function returns a linear list of the handlers in the given *scriptObject*. Each handler name is presented as a symbol in the list. This function is useful for debugging movies.

Note that you cannot get the handlers of a script cast member directly. You have to get them via the `script` property of the member.

### Examples

This statement displays the list of handlers in the child object `RedCar` in the Message window:

```
put RedCar.handlers()
-- [#accelerate, #turn, #stop]
```

This statement displays the list of handlers in the parent script member `CarParentScript` in the Message window:

```
put member("CarParentScript").script.handlers()  
-- [#accelerate, #turn, #stop]
```

**See also**

`handler()`, `script`

## height

**Syntax**

```
member(whichCastMember).height  
the height of member whichCastMember  
imageObject.height  
sprite(whichSprite).height  
the height of sprite whichSprite
```

**Description**

Cast member, image object and sprite property; for vector shape, Flash, animated GIF, bitmap, and shape cast members, determines the height, in pixels, of the cast member displayed on the Stage.

- For cast members, works with bitmap and shape cast members only. This property can be tested but not set.
- For image objects, this property can be tested but not set.
- For sprites, setting the sprite's height automatically sets the sprite's `stretch` property to `TRUE`. For the value set by Lingo to last beyond the current sprite, the sprite must be a puppet. This property can be tested and set.

**Examples**

This statement assigns the height of cast member `Headline` to the variable `vHeight`:

```
vHeight = member("Headline").height
```

This statement sets the height of sprite 10 to 26 pixels:

```
sprite(10).height = 26
```

This statement assigns the height of sprite (`i + 1`) to the variable `vHeight`:

```
vHeight = sprite(i + 1).height
```

**See also**

`height`, `rect (sprite)`, `width`

## height (3D)

**Syntax**

```
member(whichCastmember).modelResource(whichModelResource).height  
member(whichCastmember).texture(whichTexture).height
```

**Description**

3D `#box` model resource, `#cylinder` model resource, and texture property; indicates the height of the object.

The height of a `#box` or `#cylinder` model resource is measured in world units and can be tested and set. The default value for this property is 50.

The height of a texture is measured in pixels and can be tested but not set. The height of the texture is rounded from the height of the source of the texture to the nearest power of 2.

#### Examples

This statement sets the height of the model resource named Tower to 225.0 world units:

```
member("3D World").modelResource("Tower").height = 225.0
```

This statement shows that the height of the texture named Marsmap is 512 pixels.

```
put member("scene").texture("Marsmap").height
-- 512
```

#### See also

length (3D), width (3D)

## heightVertices

#### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
heightVertices
```

#### Description

3D #box model resource property; indicates the number of mesh vertices along the height of the box. Increasing this value increases the number of faces, and therefore the fineness, of the mesh.

The height of a box is measured along its Y axis.

Set the `renderStyle` property of a model's shader to `#wire` to see the faces of the mesh of the model's resource. Set the `renderStyle` property to `#point` to see just the vertices of the mesh.

The value of this property must be greater than or equal to 2. The default value is 4.

#### Example

The following statement sets the `heightVertices` property of the model resource named Tower to 10. Nine polygons will be used to define the geometry of the model resource along its Z axis; therefore, there will be ten vertices.

```
member("3D World").modelResource("Tower").heightVertices = 10
```

#### See also

height (3D)

## highlightPercentage

#### Syntax

```
member(whichCastmember).model(whichModel).toon.highlightPercentage
member(whichCastmember).model(whichModel).shader.highlight\
Percentage
member(whichCastmember).shader(whichShader).highlightPercentage
```

#### Description

3D toon modifier and #painter shader property; indicates the percentage of available colors that are used in the area of the model's surface where light creates highlights.

The range of this property is 0 to 100, and the default value is 50.

The number of colors used by the toon modifier and #painter shader for a model is determined by the `colorSteps` property of the model's toon modifier or #painter shader.



### Example

The following statement sets the `highlightPercentage` property of the `toon` modifier for the model named `Sphere` to 50. Half of the colors available to the `toon` modifier for this model will be used for the highlight area of the model's surface.

```
member("shapes").model("Sphere").toon.highlightPercentage = 50
```

### See also

`highlightStrength`, `brightness`

## highlightStrength

### Syntax

```
member(whichCastmember).model(whichModel).toon.highlightStrength  
member(whichCastmember).model(whichModel).shader.highlightStrength  
member(whichCastmember).shader(whichShader).highlightStrength
```

### Description

3D `toon` modifier and `#painter` shader property; indicates the brightness of the area of the model's surface where light creates highlights.

The default value of this property is 1.0.

### Example

The following statement sets the `highlightStrength` property of the `toon` modifier for the model named `Teapot` to 0.5. The model's highlights will be moderately bright.

```
member("shapes").model("Teapot").toon.highlightStrength = 0.5
```

### See also

`highlightPercentage`, `brightness`

## hilite (command)

### Syntax

```
fieldChunkExpression.hilite()  
hilite fieldChunkExpression
```

### Description

Command; highlights (selects) in the field sprite the specified chunk, which can be any chunk that Lingo lets you define, such as a character, word, or line. On the Macintosh, the highlight color is set in the Color control panel.

### Examples

This statement highlights the fourth word in the field cast member `Comments`, which contains the string `Thought for the Day`:

```
member("Comments").word[4].hilite()
```

This statement causes highlighted text within the sprite for field `myRecipes` to be displayed without highlighting:

```
myLineCount = member("myRecipes").line.count  
member("myRecipes").line[myLineCount + 1].hilite()
```

### See also

`char...of`, `item...of`, `line...of`, `word...of`, `delete`, `mouseChar`, `mouseLine`, `mouseWord`, `field`, `selection()` (function), `selEnd`, `selStart`

## hilite (cast member property)

### Syntax

`member(whichCastMember).hilite`  
the `hilite` of member *whichCastMember*

### Description

Cast member property; determines whether a check box or radio button created with the button tool is selected (TRUE) or not (FALSE, default).

If *whichCastMember* is a string, it specifies the cast member name. If it is an integer, *whichCastMember* specifies the cast member number.

This property can be tested and set.

### Examples

This statement checks whether the button named Sound on is selected and, if it is, turns sound channel 1 all the way up:

```
if member("Sound on").hilite = TRUE then sound(1).volume = 255
```

This statement uses Lingo to select the button cast member powerSwitch by setting the `hilite` member property for the cast member to TRUE:

```
member("powerSwitch").hilite = TRUE
```

### See also

`checkBoxAccess`, `checkBoxType`

## hitTest()

### Syntax

`sprite(whichFlashSprite).hitTest(point)`  
`hitTest(sprite whichFlashSprite, point)`

### Description

Function; indicates which part of a Flash movie is directly over a specific Director Stage location. The Director Stage location is expressed as a Director point value: for example, `point(100,50)`. The `hitTest` function returns these values:

- `#background`—The specified Stage location falls within the background of the Flash movie sprite.
- `#normal`—The specified Stage location falls within a filled object.
- `#button`—The specified Stage location falls within the active area of a button.
- `#editText`—The specified Stage location falls within a Flash editable text field.

### Example

This frame script checks to see if the mouse is currently located over a button in a Flash movie sprite in channel 5 and, if it is, the script sets a text field used to display a status message:

```
on exitFrame
  if sprite(5).hitTest(the mouseLoc) = #button then
    member("Message Line").text = "Click here to play the movie."
    updateStage
  else
    member("Message Line").text = ""
  end if
  go the frame
end
```

## hither

### Syntax

```
member(whichCastmember).camera(whichCamera).hither
sprite(whichSprite).camera{(index)}.hither
```

### Description

3D camera property; indicates the distance in world units from the camera beyond which models are drawn. Objects closer to the camera than `hither` are not drawn.

The value of this property must be greater than or equal to 1.0 and has a default value of 5.0.

### Example

The following statement sets the `hither` property of camera 1 to 1000. Models closer than 1000 world units from the camera will not be visible.

```
member("SolarSystem").camera[1].hither = 1000
```

### See also

`yon`

## HMStoFrames()

### Syntax

```
HMStoFrames(hms, tempo, dropFrame, fractionalSeconds)
```

### Description

Function; converts movies measured in hours, minutes, and seconds to the equivalent number of frames or converts a number of hours, minutes, and seconds into time if you set the *tempo* argument to 1 (1 frame = 1 second).

- *hms*—String expression that specifies the time in the form sHH:MM:SS.FFD, where:

---

s	A character is used if the time is less than zero, or a space if the time is greater than or equal to zero.
HH	Hours.
MM	Minutes.
SS	Seconds.
FF	Indicates a fraction of a second if <i>fractionalSeconds</i> is TRUE or frames if <i>fractionalSeconds</i> is FALSE.
D	A d is used if <i>dropFrame</i> is TRUE, or a space if <i>dropFrame</i> is FALSE.

---

- *tempo*—Specifies the tempo in frames per second.
- *dropFrame*—Logical expression that determines whether the frame is a drop frame (TRUE) or not (FALSE). If the string *hms* ends in a *d*, the time is treated as a drop frame, regardless of the value of *dropFrame*.
- *fractionalSeconds*—Logical expression that determines the meaning of the numbers after the seconds; they can be either fractional seconds rounded to the nearest hundredth of a second (TRUE) or the number of residual frames (FALSE).

### Examples

This statement determines the number of frames in a 1-minute, 30.1-second movie when the tempo is 30 frames per second. Neither the *dropFrame* nor *fractionalSeconds* arguments is used.

```
put HMStoFrames(" 00:01:30.10 ", 30, FALSE, FALSE)
-- 2710
```

This statement converts 600 seconds into minutes:

```
>> put framesToHMS(600, 1,0,0)
>> -- " 00:10:00.00 "
```

This statement converts an hour and a half into seconds:

```
>> put HMStoFrames("1:30:00", 1,0,0)
>> -- 5400
```

### See also

`framesToHMS()`

## hold

### Syntax

```
sprite(whichFlashSprite).hitTest(point)
hold sprite whichFlashSprite
```

### Description

Flash command; stops a Flash movie sprite that is playing in the current frame, but any audio continues to play.

### Example

This frame script holds the Flash movie sprites playing in channels 5 through 10 while allowing the audio for these channels to continue playing:

```
on enterFrame
  repeat with i = 5 to 10
    sprite(i).hold()
  end repeat
end
```

### See also

`movieRate`, `pause` (movie playback)

## hotSpot

### Syntax

`member(whichCursorCastMember).hotspot`  
the hotspot of member *whichCursorCastMember*

### Description

Cursor cast member property; specifies the horizontal and vertical point location of the pixel that represents the hotspot within the animated color cursor cast member `whichCursorCastMember`. Director uses this point to track the cursor's position on the screen (for example, when it returns the values for the Lingo functions `mouseH` and `mouseV`) and to determine where a rollover (signaled by the Lingo message `mouseEnter`) occurs.

The upper left corner of a cursor is `point(0,0)`, which is the default `hotSpot` value. Trying to set a point outside the bounds of the cursor produces an error. For example, setting the hotspot of a 16-by-16-pixel cursor to `point(16,16)` produces an error (because the starting point is 0,0, not 1,1).

This property can be tested and set.

### Example

This handler sets the hotspot of a 32-by-32-pixel cursor (whose cast member number is stored in the variable `cursorNum`) to the middle of the cursor:

```
on startMovie
  member(cursorNum).hotSpot = point(16,16)
end
```

## hotSpotEnterCallback

### Syntax

`sprite(whichQTVRSprite).hotSpotEnterCallback`  
the `hotSpotEnterCallback` of sprite *whichQTVRSprite*

### Description

QuickTime VR sprite property; contains the name of the handler that runs when the cursor enters a QuickTime VR hot spot that is visible on the Stage. The QuickTime VR sprite receives the message first. The message has two arguments: the `me` parameter and the ID of the hot spot that the cursor entered.

To clear the callback, set this property to 0.

To avoid a performance penalty, set a callback property only when necessary.

This property can be tested and set.

### See also

`hotSpotExitCallback`, `nodeEnterCallback`, `nodeExitCallback`, `triggerCallback`

## hotSpotExitCallback

### Syntax

`sprite(whichQTVRSprite).hotSpotExitCallback`  
the `hotSpotExitCallback` of `sprite` *whichQTVRSprite*

### Description

QuickTime VR sprite property; contains the name of the handler that runs when the cursor leaves a QuickTime VR hot spot that is visible on the Stage. The QuickTime VR sprite receives the message first. The message has two arguments: the `me` parameter and the ID of the hot spot that the cursor entered.

To clear the callback, set this property to 0.

To avoid a performance penalty, set a callback property only when necessary.

This property can be tested and set.

### See also

`hotSpotEnterCallback`, `nodeEnterCallback`, `nodeExitCallback`, `triggerCallback`

## HTML

### Syntax

`member(whichMember).HTML`

### Description

Cast member property; accesses text and tags that control the layout of the text within an HTML-formatted text cast member.

This property can be tested and set.

### Example

This statement displays in the message window the HTML formatting information embedded in the text cast member Home Page:

```
put member("Home Page").HTML
```

### See also

`importFileInto`, `RTF`

## hyperlink

### Syntax

`chunkExpression.hyperlink`

### Description

Text cast member property; returns the hyperlink string for the specified chunk expression in the text cast member.

This property can be both tested and set.

When retrieving this property, the link containing the first character of *chunkExpression* is used.

Hyperlinks may not overlap. Setting a hyperlink over an existing link, even partially over it), replaces the initial link with the new one.

Setting a hyperlink to an empty string removes it.

### Example

The following handler creates a hyperlink in the first word of text cast member “MacroLink”. The text is linked to Macromedia’s website.

```
on startMovie
  member("MacroLink").word[1].hyperlink = \
    "http://www.macromedia.com"
end
```

### See also

hyperlinkRange, hyperlinkState

## on hyperlinkClicked

### Syntax

```
on hyperlinkClicked me, data, range
  statement(s)
end
```

### Description

System message and event handler; used to determine when a hyperlink is actually clicked.

This event handler has the following parameters:

- *me*—Used in a behavior to identify the sprite instance
- *data*—The hyperlink data itself; the string entered in the Text Inspector when editing the text cast member
- *range*—The character range of the hyperlink in the text (It’s possible to get the text of the range itself by using the syntax member Ref.char[range[1]..range[2]])

This handler should be attached to a sprite as a behavior script. Avoid placing this handler in a cast member script.

### Example

This behavior shows a link examining the hyperlink that was clicked, jump to a URL if needed, then output the text of the link itself to the message window:

```
property spriteNum
on hyperlinkClicked me, data, range
  if data starts "http://" then
    goToNetPage(data)
  end if
  currentMember = sprite(spriteNum).member
  anchorString = currentMember.char[range[1]..range[2]]
  put "The hyperlink on"&&anchorString&&"was just clicked."
end
```

## hyperlinkRange

### Syntax

```
chunkExpression.hyperlinkRange
```

### Description

Text cast member property; returns the range of the hyperlink that contains the first character of the chunk expression.

This property can be tested but not set.

Like `hyperLink` and `hyperLinkState`, the returned range of the link contains the first character of *chunkExpression*.

**See also**

`hyperlink`, `hyperlinkState`

## hyperlinks

**Syntax**

*chunkExpression*.`hyperlinks`

**Description**

Text cast member property; returns a linear list containing all the hyperlink ranges for the specified chunk of a text cast member. Each range is given as a linear list with two elements, one for the starting character of the link and one for the ending character.

**Example**

This statement returns all the links for the text cast member *Glossary* to the message window:

```
put member("Glossary").hyperlinks
-- [[3, 8], [10, 16], [41, 54]]
```

## hyperlinkState

**Syntax**

*textChunk*.`hyperlinkState`

**Description**

Text cast member property; contains the current state of the hyperlink. Possible values for the state are: `#normal`, `#active`, and `#visited`.

This property can be tested and set.

Like `hyperLink` and `hyperLinkRange`, the returned range of the link contains the first character of *chunkExpression*.

**Example**

The following handler checks to see if the hyperlink clicked is a web address. If it is, the state of the hyperlink text state is set to `#visited`, and the movie branches to the web address.

```
property spriteNum
on hyperlinkClicked me, data, range
  if data starts "http://" then
    currentMember = sprite(spriteNum).member
    currentMember.word[4].hyperlinkState = #visited
    gotoNetPage(data)
  end if
end
```

**See also**

`hyperlink`, `hyperlinkRange`



# identity()

## Syntax

```
member(whichCastmember).model(whichModel).transform.identity()  
member(whichCastmember).group(whichGroup).transform.identity()  
member(whichCastmember).camera(whichCamera).transform.identity()  
sprite(whichSprite).camera{(index)}.transform.identity()  
member(whichCastmember).light(whichLight).transform.identity()  
transformReference.identity()
```

## Description

3D command; sets the transform to the identity transform, which is  
`transform(1.0000,0.0000,0.0000,0.0000, 0.0000,1.0000,0.0000,0.0000,  
0.0000,0.0000,1.0000,0.0000, 0.0000,0.0000,0.0000,1.0000).`

The position property of the identity transform is `vector(0, 0, 0).`

The rotation property of the identity transform is `vector(0, 0, 0).`

The scale property of the identity transform is `vector(1, 1, 1).`

The identity transform is parent-relative.

## Example

This statement sets the transform of the model named Box to the identity transform:

```
member("3d world").model("Box").transform.identity()
```

## See also

```
transform (property),getWorldTransform()
```

# on idle

## Syntax

```
on idle  
    statement(s)  
end
```

## Description

System message and event handler; contains statements that run whenever the movie has no other events to handle and is a useful location for Lingo statements that you want to execute as frequently as possible, such as statements that update values in global variables and displays current movie conditions.

Because statements in `on idle` handlers run frequently, it is good practice to avoid placing Lingo that takes a long time to process in an `on idle` handler.

It is often preferable to put `on idle` handlers in frame scripts instead of movie scripts to take advantage of the `on idle` handler only when appropriate.

Director can load cast members from an internal or external cast during an `idle` event. However, it cannot load linked cast members during an `idle` event.

The `idle` message is only sent to frame scripts and movie scripts.

### Example

This handler updates the time being displayed in the movie whenever there are no other events to handle:

```
on idle
    member("Time").text = the short time
end idle
```

### See also

`idleHandlerPeriod`

## idleHandlerPeriod

### Syntax

the `idleHandlerPeriod`

### Description

Movie property; determines the maximum number of ticks that passes until the movie sends an `idle` message. The default value is 1, which tells the movie to send `idle` handler messages no more than 60 times per second.

When the playhead enters a frame, Director starts a timer, repaints the appropriate sprites on the Stage, and issues an `enterFrame` event. Then, if the amount of time set for the tempo has elapsed, Director generates an `exitFrame` event and goes to the next specified frame; if the amount of time set for this frame hasn't elapsed, Director waits until the time runs out and periodically generates an `idle` message. The amount of time between `idle` events is determined by `idleHandlerPeriod`.

Possible settings for `idleHandlerPeriod` are:

- 0—As many idle events as possible
- 1—Up to 60 per second
- 2—Up to 30 per second
- 3—Up to 20 per second
- $n$ —Up to  $60/n$  per second

The number of `idle` events per frame also depends on the frame rate of the movie and other activity, including whether Lingo scripts are executing. If the tempo is 60 frames per second (fps) and the `idleHandlerPeriod` value is 1, one `idle` event per frame occurs. If the tempo is 20 fps, three `idle` events per frame occur. Idle time results from Director doesn't have a current task to perform and cannot generate any events.

In contrast, if the `idleHandlerPeriod` property is set to 0 and the tempo is very low, thousands of `idle` events can be generated.

The default value for this property is 1, which differs from previous versions in which it defaulted to 0.

### Example

The following statement causes the movie to send an `idle` message a maximum of once per second:

```
the idleHandlerPeriod = 60
```

### See also

`idleLoadDone()`, `idleLoadMode`, `idleLoadTag`, `idleReadChunkSize`

## idleLoadDone()

### Syntax

```
idleLoadDone(loadTag)
```

### Description

Function; reports whether all cast members with the given tag have been loaded (TRUE) or are still waiting to be loaded (FALSE).

### Example

This statement checks whether all cast members whose load tag is 20 have been loaded and then plays the movie Kiosk if they are:

```
if idleLoadDone(20) then play movie("on idle")
```

### See also

idleHandlerPeriod, idleLoadMode, idleLoadPeriod, idleLoadTag, idleReadChunkSize

## idleLoadMode

### Syntax

```
the idleLoadMode
```

### Description

System property; determines when the `preLoad` and `preLoadMember` commands try to load cast members during idle periods according to the following values:

- 0—Does not perform idle loading
- 1—Performs idle loading when there is free time between frames
- 2—Performs idle loading during `idle` events
- 3—Performs idle loading as frequently as possible

The `idleLoadMode` system property performs no function and works only in conjunction with the `preLoad` and `preLoadMember` commands.

Cast members that were loaded using idle loading remain compressed until the movie uses them. When the movie plays back, it may have noticeable pauses while it decompresses the cast members.

### Example

This statement causes the movie to try as frequently as possible to load cast members designated for preloading by the `preLoad` and `preLoadMember` commands:

```
the idleLoadMode = 3
```

### See also

idleHandlerPeriod, idleLoadDone(), idleLoadPeriod, idleLoadTag, idleReadChunkSize

## idleLoadPeriod

### Syntax

the idleLoadPeriod

### Description

System property; determines the number of ticks that Director waits before trying to load cast members waiting to be loaded. The default value for idleLoadPeriod is 0, which instructs Director to service the load queue as frequently as possible.

### Example

This statement instructs Director to try loading every 1/2 second (30 ticks) any cast members waiting to be loaded:

```
set the idleLoadPeriod = 30
```

### See also

idleLoadDone(), idleLoadMode, idleLoadTag, idleReadChunkSize

## idleLoadTag

### Syntax

the idleLoadTag

### Description

System property; identifies or tags with a number the cast members that have been queued for loading when the computer is idle. The idleLoadTag is a convenience that identifies the cast members in a group that you want to preload.

The property can be tested and set using any number that you choose.

### Example

This statement makes the number 10 the idle load tag:

```
the idleLoadTag = 10
```

### See also

idleHandlerPeriod, idleLoadDone(), idleLoadMode, idleLoadPeriod, idleReadChunkSize

## idleReadChunkSize

### Syntax

the idleReadChunkSize

### Description

System property; determines the maximum number of bytes that Director can load when it attempts to load cast members from the load queue. The default value is 32K.

This property can be tested and set.

### Example

This statement specifies that 500K is the maximum number of bytes that Director can load in one attempt at loading cast members in the load queue:

```
the idleReadChunkSize = 500 * 1024
```

### See also

idleHandlerPeriod, idleLoadDone(), idleLoadMode, idleLoadPeriod, idleLoadTag

# if

## Syntax

```
if logicalExpression then statement
if logicalExpression then statement
else statement
end if
if logicalExpression then
    statement(s)
end if
if logicalExpression then
    statement(s)
else
    statement(s)
end if
if logicalExpression1 then
    statement(s)
else if logicalExpression2 then
    statement(s)
else if logicalExpression3 then
    statement(s)
end if
if logicalExpression1 then
    statement(s)
else logicalExpression2
end if
```

## Description

**Keyword;** *if...then* structure that evaluates the logical expression specified by *logicalExpression*.

- If the condition is TRUE, Lingo executes the *statement(s)* that follow *then*.
- If the condition is FALSE, Lingo executes the *statement(s)* following *else*. If no statements follow *else*, Lingo exits the *if...then* structure.
- All parts of the condition must be evaluated; execution does not stop at the first condition that is met or not met. Thus, faster code may be created by nesting *if...then* statements on separate lines instead of placing them all on the first line to be evaluated.

When the condition is a property, Lingo automatically checks whether the property is TRUE. You don't need to explicitly add the phrase `= TRUE` after the property.

The *else* portion of the statement is optional. To use more than one *then-statement* or *else-statement*, you must end with the form `end if`.

The *else* portion always corresponds to the previous *if* statement; thus, sometimes you must include an *else nothing* statement to associate an *else* keyword with the proper *if* keyword.

**Note:** A quick way to determine in the script window if a script is paired properly is to press Tab. This forces Director to check the open Script window and show the indentation for the contents. Any mismatches will be immediately apparent.

## Examples

This statement checks whether the carriage return was pressed and then continues if it was:

```
if the key = RETURN then go the frame + 1
```

This handler checks whether the Command and Q keys were pressed simultaneously and, if so, executes the subsequent statements:

```
on keyDown
  if (the commandDown) and (the key = "q") then
    cleanUp
    quit
  end if
end keyDown
```

Compare the following two constructions and the performance results. The first construction evaluates both conditions, and so must determine the time measurement, which may take a while. The second construction evaluates the first condition; the second condition is checked only if the first condition is TRUE.

```
spriteUnderCursor = rollover()
if (spriteUnderCursor > 25) AND MeasureTimeSinceIStarted() then
  alert "You found the hidden treasure!"
end if
```

The alternate, and faster, construction would be as follows:

```
spriteUnderCursor = rollover()
if (spriteUnderCursor > 25) then
  if MeasureTimeSinceIStarted() then
    alert "You found the hidden treasure!"
  end if
end if
```

#### See also

case

## ignoreWhiteSpace()

### Syntax

```
XMLparserObject.ignoreWhiteSpace(trueOrFalse)
```

### Description

XML Command; specifies whether the parser should ignore or retain white space when generating a Lingo list. When `ignoreWhiteSpace()` is set to TRUE (the default), the parser ignores white space. When set to FALSE, the parser will retain white space and treat it as actual data.

If an element has separate beginning and ending tags, such as `<sample> </sample>`, character data within the element will be ignored if, and only if, it is composed of white space only. If there is any non-white space, or if `ignoreWhiteSpace()` is set to FALSE, there will be a CDATA node with the exact text, including any white space.

### Examples

These Lingo statements leave `ignoreWhiteSpace()` set to the default of TRUE and parse the given XML into a list. Note that the element `<sample>` has no children in the list.

```
XMLtext = "<sample> </sample>"
parserObj.parseString(XMLtext)
theList = parserObj.makelist()
put theList
-- ["ROOT OF XML DOCUMENT": ["!ATTRIBUTES": [:], "sample": ["!ATTRIBUTES":
[:]]]]
```

These Lingo statements set `ignoreWhiteSpace()` to `FALSE` and then parse the given XML into a list. Note that the element `<sample>` now has a child containing one space character.

```
XMLtext = "<sample> </sample>"
parserObj.ignorewhitespace(FALSE)
parserObj.parseString(XMLtext)
theList = parserObj.makeList()
put theList
-- ["ROOT OF XML DOCUMENT": ["!ATTRIBUTES": [:], "sample": ["!ATTRIBUTES":
  [:], "!CHARDATA": " "]]]
```

These Lingo statements leave `ignoreWhiteSpace()` set to the default of `TRUE` and parse the given XML. Note that there is only one child node of the `<sample>` tag and only one child node of the `<sub>` tag.

```
XMLtext = "<sample> <sub> phrase 1 </sub></sample>"
parserObj.parseString(XMLtext)
theList = parserObj.makeList()
put theList
-- ["ROOT OF XML DOCUMENT": ["!ATTRIBUTES": [:], "sample": ["!ATTRIBUTES":
  [:], "sub": ["!ATTRIBUTES": [:], "!CHARDATA": " phrase 1 "]]]]]
```

These Lingo statements set `ignoreWhiteSpace()` to `FALSE` and parse the given XML. Note that there are now two child nodes of the `<sample>` tag, the first one being a single space character.

```
XMLtext = "<sample> <sub> phrase 1 </sub></sample>"
gparser.ignoreWhiteSpace(FALSE)
gparser.parseString(XMLtext)
theList = gparser.makeList()
put theList
-- ["ROOT OF XML DOCUMENT": ["!ATTRIBUTES": [:], "sample": ["!ATTRIBUTES":
  [:], "!CHARDATA": " ", "sub": ["!ATTRIBUTES": [:], "!CHARDATA": " phrase 1
  "]]]]]
```

## ilk()

### Syntax

```
ilk(object)
ilk(object, type)
```

### Description

Function; indicates the type of an object.

- The syntax `ilk(object)` returns a value indicating the type of an object. If the object is a list, `ilk(object)` returns `#list`; if the object is a property list, `ilk(object)` returns `#proplist`.
- The syntax `ilk(object, type)` compares the object represented by `object` to the specified type. If the object is of the specified type, the `ilk()` function returns `TRUE`. If the object is not of the specified type, the `ilk()` function returns `FALSE`.

The following table shows the return value for each type of object recognized by `ilk()`:

Type of Object	ilk(Object) returns	ilk(Object, Type) returns 1 only if Type =	Example
linear list	<code>#list</code>	<code>#list</code> or <code>#linearlist</code>	<code>ilk ([1,2,3])</code>
property list	<code>#proplist</code>	<code>#list</code> or <code>#proplist</code>	<code>ilk ([#his: 1234, #hers: 7890])</code>
integer	<code>#integer</code>	<code>#integer</code> or <code>#number</code>	<code>ilk (333)</code>
float	<code>#float</code>	<code>#float</code> or <code>#number</code>	<code>ilk (123.456)</code>

Type of Object	ilk(Object) returns	ilk(Object, Type) returns 1 only if Type =	Example
string	#string	#string	ilk ("asdf")
rect	#rect	#rect or #list	ilk (sprite(1).rect)
point	#point	#point or #list	ilk (sprite(1).loc)
color	#color	#color	ilk (sprite(1).color)
date	#date	#date	ilk (the systemdate)
symbol	#symbol	#symbol	ilk (#hello)
void	#void	#void	ilk (void)
picture	#picture	#picture	ilk (member (2).picture)
parent script instance	#instance	#object	ilk (new (script "blahblah"))
xtra instance	#instance	#object	ilk (new (xtra "fileio"))
member	#member	#object or #member	ilk (member 1)
xtra	#xtra	#object or #xtra	ilk (xtra "fileio")
script	#script	#object or #script	ilk (script "blahblah")
castlib	#castlib	#object or #castlib	ilk (castlib 1)
sprite	#sprite	#object or #sprite	ilk (sprite 1)
sound	#instance or #sound (when Sound Control Xtra is not present)	#instance or #sound	ilk (sound "yaddayadda")
window	#window	#object or #window	ilk (the stage)
media	#media	#object or #media	ilk (member (2).media)
timeout	#timeout	#object or #timeout	ilk (timeOut("intervalTimer"))
image	#image	#object or #image	ilk ((the stage).image)

### Examples

The following ilk statement identifies the type of the object named Bids:

```
Bids = [:]
put ilk( Bids )
-- #proplist
```

The following ilk statement tests whether the variable Total is a list and displays the result in the Message window:

```
Total = 2+2
put ilk( Total, #list )
-- 0
```

In this case, since the variable Total is not a list, the Message window displays 0, which is the numeric equivalent of FALSE.



The following example tests a variable named `myVariable` and verifies that it is a date object before displaying it in the Message window:

```
myVariable = the systemDate
if ilk(myVariable, #date) then put myVariable
-- date( 1999, 2, 19 )
```

## ilk (3D)

### Syntax

```
ilk(object)
ilk(object, type)
object.ilk
object.ilk(type)
```

### Description

Lingo function; indicates the type of an object.

- The syntax `ilk(object)` and `object.ilk` return a value indicating the type of object. If the object is a model, `ilk(object)` returns `#model`; if the object is a motion, `ilk(object)` returns `#motion`. See the following table for a complete list of values returned by 3D objects.
- The syntax `ilk(object, type)` and `object.ilk(type)` compare the object represented by the object to the specified type. If the object is of the specified type, the `ilk()` function returns `TRUE`. If the object is not of the specified type, the `ilk()` function returns `FALSE`.

The following table shows the return value for each type of 3D object recognized by `ilk()`. See the main Lingo Dictionary for a list of return values of non-3D objects which are not discussed in this dictionary.

Type of object	ilk(object) returns	ilk(object, Type) if only Type =
render services	<code>#renderer</code>	<code>#renderer</code>
model resource	<code>#modelresource</code> , <code>#plane</code> , <code>#box</code> , <code>#sphere</code> , <code>#cylinder</code> , <code>#particle</code> , <code>#mesh</code>	Same as <code>ilk(object)</code> , except for <code>#modelresource</code> which is the <code>ilk</code> of resources generated by an imported W3D file
model	<code>#model</code>	<code>#model</code>
motion	<code>#motion</code>	<code>#motion</code> or <code>#list</code>
shader	<code>#shader</code>	<code>#shader</code> or <code>#list</code>
texture	<code>#texture</code>	<code>#texture</code> or <code>#list</code>
group	<code>#group</code>	<code>#group</code>
camera	<code>#camera</code>	<code>#camera</code>
collision data	<code>#collisiondata</code>	<code>#collisiondata</code>
vector	<code>#vector</code>	<code>#vector</code>
transform	<code>#transform</code>	<code>#transform</code>

### Examples

This statement shows that `MyObject` is a motion object:

```
put MyObject.ilk
-- #motion
```

The following statement tests whether `MyObject` is a motion object. The return value of 1 shows that it is.

```
put MyObject.ilk(#motion)
-- 1
```

### See also

`tweenMode`

## image

### Syntax

```
whichMember.image
(the stage).image
window(windowName).image
```

### Description

This property refers to the image object of a bitmap or text cast member, of the Stage, or of a window. You can get or set a cast member's image, but you can only get the image of the Stage or a window.

Setting a cast member's image property immediately changes the contents of the member. However, when you get the image of a member or window, Director creates a reference to the image of the specified member or window. If you make changes to the image, the contents of the cast member or window change immediately.

If you plan to make a lot of changes to an item's image property, it is faster to copy the item's image property into a new image object using the `duplicate()` function, apply your changes to the new image object, and then set the original item's image to the new image object. For non-bitmap members, it is always faster to use the `duplicate()` function.

### Examples

This statement puts the image of cast member `originalFlower` into cast member `newFlower`:

```
member("newFlower").image = member("originalFlower").image
```

These statements place a reference to the image of the stage into the variable `myImage` and then put that image into cast member `flower`:

```
myImage=(the stage).image
member("flower").image = myImage
```

### See also

`setPixel()`, `draw()`, `image()`, `fill()`, `duplicate()` (image function), `copyPixels()`

## image()

### Syntax

```
image(width, height, bitDepth {, alphaDepth} {, paletteSymbolOrMember})
```

### Description

Function; creates and returns a new image object of the dimensions specified by *width*, *height*, and *bitDepth*, with optional *alphaDepth* and *paletteObject* values.

The *bitDepth* can be 1, 2, 4, 8, 16, or 32. The *alphaDepth*, if given, is used only for 32-bit images and must be 0 or 8. The *paletteObject*, if given, is used only for 2-, 4-, and 8-bit images and can be either a palette symbol, such as `#grayscale`, or a palette cast member. If no palette is specified, the movie's default palette is used.

Image objects created with `image()` are independent and do not refer to any cast member or window.

To see an example of `image()` used in a completed movie, see the Imaging movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

These statements create a 200 x 200 pixel, 8-bit image object and fill the image object with red:

```
redSquare = image(200, 200, 8)
redSquare.fill(0, 0, 200, 200, rgb(255, 0, 0))
```

### See also

`palette`, `image`, `duplicate()` (image function), `fill()`

## image (RealMedia)

### Syntax

```
sprite(whichSprite).image
member(whichCastmember).image
```

### Description

RealMedia sprite or cast member property; returns a Lingo image object containing the current frame of the RealMedia video stream. You can use this property to map RealVideo onto a 3D model (see the example below).

### Example

This statement copies the current frame of the RealMedia cast member `Real` to the bitmap cast member `Still`:

```
member("Still").image = member("Real").image
```

This handler is called by a frame script once per Director frame. The handler creates a new texture from the image of the RealMedia cast member named Real if it is playing or paused. The new texture is then used by the shader of the model named mSphere. The texture that was used in the previous frame is deleted. Finally, the sphere is rotated by 1° around the *y*-axis. The result is a #realMedia video playing on a rotating sphere.

```
on updateShader
  if member("Real").state = 4 then
    sphereShader = member("3d").model("mSphere").shader
    tex = member("3d").newTexture("texture" & gTextureNumber, \
      #fromImageObject, member("Real").image)
    sphereShader.texture = tex
    if gTextureNumber > 1 then
      member("3d").deleteTexture("texture" & (gTextureNumber - 1))
    end if
    gTextureNumber = gTextureNumber + 1
  end if
  member("3d").model[1].transform.rotate(0, 1, 0)
end
```

## imageCompression

### Syntax

member(*whichMember*).imageCompression  
the imageCompression of member *whichMember*

### Description

This bitmap cast member property indicates the type of compression that Director will apply to the member when saving the movie in Shockwave format. This property can be tested and set, and has no effect at runtime. Its value can be any one of these symbols:

Value	Meaning
#movieSetting	Use the compression settings of the movie, as stored in the movieImageCompression property. This is the default value for image formats not restricted to standard compression (see below).
#standard	Use the Director standard internal compression format.
#jpeg	Use JPEG compression. See imageQuality.

You normally set this property in the Property inspector's Bitmap tab. However, if you want to set this property for a large number of images at once, you can set the property with a Lingo routine.

If a member doesn't support JPEG compression because it is 8-bit or lower, or if the image is linked from an external file, only #standard compression can be used. Image formats that do not support JPEG compression include GIF and 8-bit or lower images.

### Example

This statement displays the imageCompression of member Sunrise in the message window:

```
put member("Sunrise").imageCompression
-- #movieSetting
```

### See also

imageQuality, movieImageCompression, movieImageQuality

## imageEnabled

### Syntax

`sprite(whichVectorOrFlashSprite).imageEnabled`  
the `imageEnabled` of sprite *whichVectorOrFlashSprite*  
`member(whichVectorOrFlashMember).imageEnabled`  
the `imageEnabled` of member *whichVectorOrFlashMember*

### Description

Cast member property and sprite property; controls whether a Flash movie or vector shape's graphics are visible (TRUE, default) or invisible (FALSE).

This property can be tested and set.

### Example

This `beginSprite` script sets up a linked Flash movie sprite to hide its graphics when it first appears on the Stage and begins to stream into memory and saves its sprite number in a global variable called `gStreamingSprite` for use in a frame script later in the Score:

```
global gStreamingSprite
on beginSprite me
    gStreamingSprite = me.spriteNum
    sprite(gStreamingSprite).imageEnabled = FALSE
end
```

In a later frame of the movie, this frame script checks to see if the Flash movie sprite specified by the global variable `gStreamingSprite` has finished streaming into memory. If it has not, the script keeps the playhead looping in the current frame until 100% of the movie has streamed into memory. It then sets the `imageEnabled` property to TRUE so that the graphics appear and lets the playhead continue to the next frame in the Score.

```
global gStreamingSprite
on exitFrame me
    if sprite(gStreamingSprite).member.percentStreamed < 100 then
        go to frame
    else
        sprite(gStreamingSprite).imageEnabled = TRUE
        updateStage
    end if
end
```

## imageQuality

### Syntax

`member(whichMember).imageQuality`  
the `imageQuality` of member *whichMember*

### Description

This bitmap cast member property indicates the level of compression to use when the member's `imageCompression` property is set to `#jpeg`. The range of acceptable values is 0–100. Zero yields the lowest image quality and highest compression; 100 yields the highest image quality and lowest compression.

This property is settable only during authoring and only affects cast members when saving a movie in Shockwave format. The compressed image can be previewed via the Optimize in Fireworks button in the Property inspector's Bitmap tab or the Preview in Browser command in the File menu.

If an image cast member's `imageCompression` property is set to `#MovieSetting`, the movie property `movieImageQuality` is used instead of `imageQuality`.

**See also**

`imageCompression`, `movieImageCompression`, `movieImageQuality`

## immovable

**Syntax**

```
member(whichCastmember).model(whichModel).collision.immovable
```

**Description**

3D `#collision` modifier property; indicates whether a model can be moved as a result of collisions during animations. Specifying `TRUE` makes the model immovable; specifying `FALSE` allows the model to be moved. This property is useful as a way of improving performance during animation, because models that do not move do not need to be checked for collisions by Lingo.

This property has a default value of `FALSE`.

**Example**

This statement sets the `immovable` property of the `collision` modifier attached to the first model of the cast member named `Scene` to `TRUE`:

```
member("Scene").model[1].collision.immovable = TRUE
```

**See also**

`collision (modifier)`

## importFileInto

**Syntax**

```
importFileInto member whichCastMember, fileName  
importFileInto member whichCastMember of castLib whichCast, fileName  
importFileInto member whichCastMember, URL
```

**Description**

Command; replaces the content of the cast member specified by *whichCastMember* with the file specified by *fileName*.

The `importFileInto` command is useful in four situations:

- When finishing developing a movie, use it to embed media that you have kept linked and external so it can be edited during the project.
- When generating Score from Lingo during movie creation, use it to assign content to new cast members that you created.

- When downloading files from the Internet, use it to download the file at a specific URL and set the filename of linked media. (However, to import a file from a URL, it's usually more efficient and minimizes downloading to use the `preloadNetThing` command to download the file to a local disk first and then import the file from the local disk.)
- Use it to import both RTF and HTML documents into text cast members with formatting and links intact.

Use of the `importFileInto` command in projectors can quickly consume available memory, so it's best to reuse the same members for imported data if possible.

Also, the `importFileInto` command doesn't work with the Director player for Java. To change the content of a bitmap or sound cast member in a movie playing back as an applet, make the cast members linked cast members and change the cast member's `fileName` property.

**Note:** In Shockwave, you must issue a `preloadNetThing` and wait for a successful completion of the download before using `importFileInto` with the file. In Director and projectors, `importFileInto` automatically downloads the file for you.

### Example

This handler assigns a URL that contains a GIF file to the variable `tempURL` and then uses the `importFileInto` command to import the file at the URL into a new bitmap cast member:

```
on exitFrame
    tempURL = "http://www.dukeOfUrl.com/crown.gif"
    importFileInto new(#bitmap), tempURL
end
```

This statement replaces the content of the sound cast member `Memory` with the sound file `Wind`:

```
importFileInto member "Memory", "Wind.wav"
```

These statements download an external file from a URL to the Director application folder and then import that file into the sound cast member `Norma Desmond Speaks`:

```
downloadNetThing http://www.cbDeMille.com/Talkies.AIF, the \
    applicationPath&"Talkies.AIF" \
importFileInto(member "Norma Desmond Speaks", the
    applicationPath&"Talkies.AIF")
```

### See also

`downloadNetThing`, `fileName` (cast member property), `preloadNetThing()`

## in

### See also

`number (characters)`, `number (items)`, `number (lines)`, `number (words)`

## INF

### Description

Lingo return value; indicates that a specified Lingo expression evaluates as an infinite number.

### See also

`NAN`

## inflate

### Syntax

```
rectangle.Inflate(widthChange, heightChange)  
inflate (rectangle, widthChange, heightChange)
```

### Description

Command; changes the dimensions of the rectangle specified by *rectangle* relative to the center of the rectangle, either horizontally (*widthChange*) or vertically (*heightChange*).

The total change in each direction is twice the number you specify. For example, replacing *widthChange* with 15 increases the rectangle's width by 30 pixels. A value less than 0 for the horizontal or vertical dimension reduces the rectangle's size.

### Examples

This statement increases the rectangle's width by 4 pixels and the height by 2 pixels:

```
rect(10, 10, 20, 20).inflate(2, 1)  
-- rect (8, 9, 22, 21)
```

This statement decreases the rectangle's height and width by 20 pixels:

```
inflate (rect(0, 0, 100, 100), -10, -10)  
-- rect (10, 10, 90, 90)
```

## ink

### Syntax

```
sprite(whichSprite).ink  
the ink of sprite whichSprite
```

### Description

Sprite property; determines the ink effect applied to the sprite specified by *whichSprite*, as follows:

---

0-Copy	32-Blend
1-Transparent	33-Add pin
2-Reverse	34-Add
3-Ghost	35-Subtract pin
4-Not copy	36-Background transparent
5-Not transparent	37-Lightest
6-Not reverse	38-Subtract
7-Not ghost	39-Darkest
8-Matte	40-Lighten
9-Mask	41-Darken

---



For a movie that plays back as an applet, valid values for the `ink` sprite property vary for different sprites, as follows:

- For bitmap sprites, the `ink` sprite property can be 0 (Copy), 8 (Matte), 32 (Blend), or 36 (Background transparent).
- For vector shape, Flash, and shape sprites, the `ink` sprite property can be 0, 8, or 36.
- For field sprites, the `ink` sprite property can be 0 or 36. The player treats Blend and Matte inks as Background transparent.

In the case of 36 (background transparent), you select a sprite in the score and select a transparency color from the background color box in the Tools window. You can also do this by setting the `backColor` property.

If you set this property within a script while the playhead is not moving, be sure to use the `updateStage` command to redraw the Stage. If you change several sprite properties—or several sprites—use only one `updateStage` command at the end of all the changes.

This property can be tested and set.

### Examples

This statement changes the variable `currentInk` to the value for the ink effect of sprite (3):

```
currentInk = sprite(3).ink
```

This statement gives sprite (`i + 1`) a matte ink effect by setting the ink effect of the sprite property to 8, which specifies matte ink:

```
sprite(i + 1).ink = 8
```

### See also

`backColor`, `foreColor`

## inker (modifier)

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
    inker.inkerModifierProperty
modelResourceObjectReference.inker.inkerModifierProperty
```

### Description

3D modifier; once you have added the `#inker` modifier to a model resource (using `addModifier`) you can get and set `#inker` modifier properties.

The `#inker` modifier adds silhouettes, creases, and boundary edges to an existing model; the `#inker` properties allow you to control the definition and emphasis of these properties.

When the `#inker` modifier is used in conjunction with the `#toon` modifier, the rendered effect is cumulative and varies depending on which modifier was first applied. The list of modifiers returned by the `modifier` property will list `#inker` or `#toon` (whichever was added first), but not both. The `#inker` modifier can not be used in conjunction with the `#sds` modifier.

The `#inker` modifier has the following properties:

- `lineColor` allows you to get or set the color of lines drawn by the inker.
- `silhouettes` allows you to get or set whether lines are drawn to define the edges along the border of a model, outlining its shape.
- `creases` allows you to get or set whether lines are drawn in creases.
- `creaseAngle` allows you to get or set the sensitivity of crease angle detection for the inker.
- `boundary` allows you to get or set whether lines are drawn around the boundary of the surface.
- `lineOffset` allows you to get or set where lines are drawn relative to the surface being shaded and the camera.
- `useLineOffset` allows you to get or set whether `lineOffset` is on or off.

**Note:** For more detailed information about the following properties see the individual property entries.

**See also**

`addModifier`, `modifiers`, `toon (modifier)`, `shadowPercentage`

## inlineImeEnabled

**Syntax**

the `inlineImeEnabled`

**Description**

Global property; determines whether the Director Inline IME feature is turned on. When `TRUE`, this property allows the user to enter double-byte characters directly into the Director Text, Field, Script and Message windows on Japanese systems.

This property can be tested and set. The default value is determined by the Enable Inline IME setting in Director General Preferences.

## insertBackdrop

**Syntax**

```
sprite(whichSprite).camera{(index)}.insertBackdrop(index, \
    texture, locWithinSprite, rotation)
member(whichCastmember).camera(whichCamera).\
    insertBackdrop(index, texture, locWithinSprite, rotation)
```

**Description**

3D camera command; adds a backdrop to the camera's list of backdrops at the position indicated by the `index` parameter. The backdrop is displayed in the 3D sprite at `locWithinSprite` with the indicated rotation. The `locWithinSprite` parameter is a 2D loc measured from the upper left corner of the sprite.

**Example**

The first line of this example creates a texture called Cedar. The second line inserts that texture at the first position in the list of backdrops of the camera of sprite 5. The backdrop is positioned at the point (300, 120), measured from the upper left corner of the sprite. It is rotated 45°.

```
t1 = member("scene").texture("Cedar")
sprite(5).camera.insertBackdrop(1, t1, point(300, 120), 45)
```

**See also**

`removeBackdrop`, `bevelDepth`, `overlay`

## insertFrame

### Syntax

`insertFrame`

### Description

Command; duplicates the current frame and its content. The duplicate frame is inserted after the current frame and then becomes the current frame.

This command can be used only during a Score recording session and performs the same function as the `duplicateFrame` command.

### Example

The following handler generates a frame that has the transition cast member Fog assigned in the transition channel followed by a set of empty frames. The argument `numberOfFrames` sets the number of frames.

```
on animBall numberOfFrames
  beginRecording
    the frameTransition = member ("Fog").number
    go the frame + 1
    repeat with i = 0 to numberOfFrames
      insertFrame
    end repeat
  endRecording
end
```

## insertOverlay

### Syntax

```
sprite(whichSprite).camera{(index)}.insertOverlay(index, \
  texture, locWithinSprite, rotation)
member(whichCastmember).camera(whichCamera).\
  insertOverlay(index, texture, \
  locWithinSprite, rotation)
```

### Description

3D camera command; adds an overlay to the camera's list of overlays at the position indicated by the *index* parameter. The overlay is displayed in the 3D sprite at *locWithinSprite* with the indicated *rotation*. The *locWithinSprite* parameter is a 2D loc measured from the upper left corner of the sprite.

### Example

The first line of this example creates a texture named Cedar. The second line inserts that texture at the first position in the list of overlays of the camera of sprite 5. The overlay is positioned at the point (300, 120), measured from the upper left corner of the sprite. It is rotated 45°.

```
t1 = member("scene").texture("Cedar")
sprite(5).camera.insertOverlay(1, t1, point(300, 120), 45)
```

### See also

`removeOverlay`, `overlay`, `bevelDepth`

## inside()

### Syntax

```
point.inside(rectangle)  
inside(point, rectangle)
```

### Description

Function; indicates whether the point specified by *point* is within the rectangle specified by *rectangle* (TRUE), or outside the rectangle (FALSE).

### Example

This statement indicates whether the point Center is within the rectangle Zone and displays the result in the Message window:

```
put Center.inside(Zone)
```

### See also

```
map(), mouseH, mouseV, point()
```

## installMenu

### Syntax

```
installMenu whichCastMember
```

### Description

Command; installs the menu defined in the field cast member specified by *whichCastMember*.

These custom menus appear only while the movie is playing. To remove the custom menus, use the `installMenu` command with no argument or with 0 as the argument. This command doesn't work with hierarchical menus.

For an explanation of how menu items are defined in a field cast member, see the `menu` keyword.

Avoid changing menus many times because doing so affects system resources.

In Windows, if the menu is longer than the screen, only part of the menu appears; on the Macintosh, menus longer than the screen can scroll.

**Note:** Menus are not available in Shockwave.

### Examples

This statement installs the menu defined in field cast member 37:

```
installMenu 37
```

This statement installs the menu defined in the field cast member named Menubar:

```
installMenu member "Menubar"
```

This statement disables menus that were installed by the `installMenu` command:

```
installMenu 0
```

### See also

```
menu
```

## integer()

### Syntax

```
(numericExpression).integer  
integer(numericExpression)
```

### Description

Function; rounds the value of *numericExpression* to the nearest whole integer.

You can force an integer to be a string by using the `string()` function.

### Examples

This statement rounds off the number 3.75 to the nearest whole integer:

```
put integer(3.75)  
-- 4
```

The following statement rounds off the value in parentheses. This provides a usable value for the `locH` sprite property, which requires an integer:

```
sprite(1).locH = integer(0.333 * stageWidth)
```

### See also

`float()`, `string()`

## integerP()

### Syntax

```
expression.integerP  
(numericExpression).integerP  
integerP(expression)
```

### Description

Function; indicates whether the expression specified by *expression* can be evaluated to an integer (1 or TRUE) or not (0 or FALSE). *P* in `integerP` stands for *predicate*.

### Examples

This statement checks whether the number 3 can be evaluated to an integer and then displays 1 (TRUE) in the Message window:

```
put(3).integerP  
-- 1
```

The following statement checks whether the number 3 can be evaluated to an integer. Because 3 is surrounded by quotation marks, it cannot be evaluated to an integer, so 0 (FALSE) is displayed in the Message window:

```
put("3").integerP  
-- 0
```

This statement checks whether the numerical value of the string in field cast member Entry is an integer and if it isn't, displays an alert:

```
if field("Entry").value.integerP = FALSE then alert "Please enter an integer."
```

### See also

`floatP()`, `integer()`, `ilk()`, `objectP()`, `stringP()`, `symbolP()`

## interface()

### Syntax

```
xtra("XtraName").interface()  
interface(xtra "XtraName")
```

### Description

Function; returns a Return-delimited string that describes the Xtra and lists its methods. This function replaces the now obsolete `mMessageList` function.

### Example

This statement displays the output from the function used in the QuickTime Asset Xtra in the Message window:

```
put Xtra("QuickTimeSupport").interface()
```

## interpolate()

### Syntax

```
transform1.interpolate(transform2,percentage)
```

### Description

3D transform method; returns a copy of *transform1* created by interpolating from the position and rotation of *transform1* to the position and rotation of *transform2* by the specified percentage. The original *transform1* is not affected. To interpolate *transform1*, use `interpolateTo()`.

To interpolate by hand, multiply the difference of two numbers by the percentage. For example, interpolation from 4 to 8 by 50 percent yields 6.

### Example

In this example, `tBox` is the transform of the model named Box, and `tSphere` is the transform of the model named Sphere. The third line of the example interpolates a copy of the transform of Box halfway to the transform of Sphere.

```
tBox = member("3d world").model("Box").transform  
tSphere = member("3d world").model("Sphere").transform  
tNew = tBox.interpolate(tSphere, 50)
```

### See also

`interpolateTo()`

## interpolateTo()

### Syntax

```
transform1.interpolateTo(transform2, percentage)
```

### Description

3D transform method; modifies *transform1* by interpolating from the position and rotation of *transform1* to the position and rotation of *transform2* by the specified percentage. The original *transform1* is changed. To interpolate a copy of *transform1*, use the `interpolate()` function.

To interpolate by hand, multiply the difference of two numbers by the percentage. For example, interpolation from 4 to 8 by 50 percent yields 6.

### Example

In this example, `tBox` is the transform of the model named `Box`, and `tSphere` is the transform of the model named `Sphere`. The third line of the example interpolates the transform of `Box` halfway to the transform of `Sphere`.

```
tBox = member("3d world").model("Box").transform
tSphere = member("3d world").model("Sphere").transform
tBox.interpolateTo(tSphere, 50)
```

### See also

`interpolate()`

## intersect()

### Syntax

```
rectangle1. Intersect(rectangle2)
intersect(rectangle1, rectangle2)
```

### Description

Function; determines the rectangle formed where *rectangle1* and *rectangle2* intersect.

### Example

This statement assigns the variable `newRectangle` to the rectangle formed where `rectangle` `toolKit` intersects `rectangle Ramp`:

```
newRectangle = toolkit.intersect(Ramp)
```

### See also

`map()`, `rect()`, `union()`

## interval

### Syntax

```
member(whichCursorCastMember).interval
the interval of member whichCursorCastMember
```

### Description

Cursor cast member property; specifies the interval, in milliseconds (ms), between each frame of the animated color cursor cast member *whichCursorCastMember*. The default interval is 100 ms.

The cursor interval is independent of the frame rate set for the movie using the tempo channel or the `puppetTempo` Lingo command.

This property can be tested and set.

### Example

In this sprite script, when the animated color cursor stored in the cast member named `Butterfly` enters the sprite, the interval is set to 50 ms to speed up the animation. When the cursor leaves the sprite, the interval is reset to 100 ms to slow down the animation.

```
on mouseEnter
    member("Butterfly").interval = 50
end

on mouseLeave
    member("Butterfly").interval = 100
end
```

## into

This code fragment occurs in a number of Lingo constructs, such as `put...into`.

## inverse()

### Syntax

```
member(whichCastmember).model(whichModel).transform.inverse()  
member(whichCastmember).group(whichGroup).transform.inverse()  
member(whichCastmember).camera(whichCamera).transform.inverse()  
sprite(whichSprite).camera[(index)].transform.inverse()  
member(whichCastmember).light(whichLight).transform.inverse()  
transformReference.inverse()
```

### Description

3D transform method; returns a copy of the transform with its position and rotation properties inverted.

This method does not change the original transform. To invert the original transform, use the `invert()` function.

### Example

This statement inverts a copy of the transform of the model named Chair:

```
boxInv = member("3d world").model("Chair").transform.inverse()
```

### See also

`invert()`

## invert()

### Syntax

```
member(whichCastmember).model(whichModel).transform.invert()  
member(whichCastmember).group(whichGroup).transform.invert()  
member(whichCastmember).camera(whichCamera).transform.invert()  
sprite(whichSprite).camera[(index)].transform.invert()  
member(whichCastmember).light(whichLight).transform.invert()  
transformReference.invert()
```

### Description

3D transform method; inverts the position and rotation properties of the transform.

This method changes the original transform. To invert a copy of the original transform, use the `inverse()` function.

### Example

This statement inverts the transform of the model Box:

```
member("3d world").model("Box").transform.invert()
```

### See also

`inverse()`



## invertMask

### Syntax

`member(whichQuickTimeMember).invertMask`  
the invertMask of member *whichQuickTimeMember*

### Description

QuickTime cast member property; determines whether Director draws QuickTime movies in the white pixels of the movie's mask (TRUE) or in the black pixels (FALSE, default).

This property can be tested and set.

### Example

This handler reverses the current setting of the `invertMask` property of a QuickTime movie named Starburst:

```
on toggleMask
    member("Starburst").invertMask = not member("Starburst").invertMask
end
```

### See also

`mask`

## isBusy()

### Syntax

`sound(channelNum).isBusy()`

### Description

Function; returns TRUE if sound channel *channelNum* is currently playing or pausing a sound, and FALSE if it hasn't started playing any of its queued sounds or has been stopped.

To see an example of `isBusy()` used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

The following Lingo statement checks whether a sound is playing in sound channel 1. If there is, the text of member field is set to "You should hear music now." Otherwise, the text reads "The music has ended."

```
if sound(1).isBusy() then
    member("field").text = "You should hear music now."
else
    member("field").text = "The music has ended."
end if
```

### See also

`pause()` (`sound playback`), `playNext()`, `queue()`, `status`, `stop()` (`sound`)

## isInWorld()

### Syntax

```
member(whichCastmember).model(whichModel).isInWorld()  
member(whichCastmember).camera(whichCamera).isInWorld()  
member(whichCastmember).light(whichLight).isInWorld()  
member(whichCastmember).group(whichGroup).isInWorld()
```

### Description

3D command; returns a value of TRUE if the parent hierarchy of the model, camera, light, or group terminates in the world. If the value of `isInWorld` is TRUE, the model, camera, light, or group functions in the 3D world of the cast member.

Models, cameras, lights, and groups can be stored in a 3D cast member but not used in the 3D world of the cast member. Use the `addToWorld` and `removeFromWorld` commands to add and remove models, cameras, lights, and groups from the 3D world of the cast member.

### Example

This statement shows that the model named Teapot exists in the 3D world of the cast member named TableScene:

```
put member("TableScene").model("Teapot").isInWorld()  
-- 1
```

### See also

`addToWorld`, `removeFromWorld`, `child`

## on isOKToAttach

### Syntax

```
on isOKToAttach me, aSpriteType, aSpriteNum
```

### Description

Built-in handler; you can add this handler to a behavior in order to check the type of sprite the behavior is being attached to and prevent the behavior from being attached to inappropriate sprite types.

When the behavior is attached to a sprite, the handler executes and Director passes to it the type of the sprite and its sprite number. The `me` argument contains a reference to the behavior that is being attached to the sprite.

This handler runs before the `on getPropertyDescriptionList` handler.

The Lingo author can check for two types of sprites. `#graphic` includes all graphic cast members, such as shapes, bitmaps, digital video, text, and so on. `#script` indicates the behavior was attached to the script channel. In this case, the `spriteNum` is 1.

For each of these sprite types, the handler must return TRUE or FALSE. A value of TRUE indicates that the behavior can be attached to the sprite. A value of FALSE prevents the behavior from being attached to the sprite.

If the behavior contains no `on isOKToAttach` handler, then the behavior can be attached to any sprite or frame.

This handler is called during the initial attachment of the behavior to the sprite or script channel and also when attaching a new behavior to a sprite using the Behavior inspector.

### Example

This statement checks the sprite type the behavior is being attached to and returns TRUE for any graphic sprite except a shape and FALSE for the script channel:

```
on isOKToAttach me, aSpriteType, aSpriteNum
  case aSpriteType of
    #graphic: -- any graphic sprite type
      return sprite(aSpriteNum).member.type <> #shape
    -- works for everything but shape cast members
    #script: --the frame script channel
      return FALSE -- doesn't work as a frame script
  end case
end
```

## isPastCuePoint()

### Syntax

```
sprite(spriteNum).isPastCuePoint(cuePointID)
isPastCuePoint(sprite(spriteNum), cuePointID)
sound(channelNum).isPastCuePoint(cuePointID)
isPastCuePoint(sound(channelNum), cuePointID)
```

### Description

Function; determines whether a sprite or sound channel has passed a specified cue point in its media. This function can be used with sound (WAV, AIFF, SND, SWA, AU), QuickTime, or Xtra files that support cue points.

Replace *spriteNum* or *channelNum* with a sprite channel or a sound channel. Shockwave Audio (SWA) sounds can appear as sprites in sprite channels, but they play sound in a sound channel. It is recommended that you refer to SWA sound sprites by their sprite channel number rather than their sound channel number.

Replace *cuePointID* with a reference for a cue point:

- If *cuePointID* is an integer, *isPastCuePoint* returns 1 if the cue point has been passed and 0 if it hasn't been passed.
- If *cuePointID* is a name, *isPastCuePoint* returns the number of cue points passed that have that name.

If the value specified for *cuePointID* doesn't exist in the sprite or sound, the function returns 0.

The number returned by *isPastCuePoint* is based on the absolute position of the sprite in its media. For example, if a sound passes cue point Main and then loops and passes Main again, *isPastCuePoint* returns 1 instead of 2.

When the result of *isPastCuePoint* is treated as a Boolean operator, the function returns TRUE if any cue points identified by *cuePointID* have passed and FALSE if no cue points are passed.

### Examples

This statement plays a sound until the third time the cue point Chorus End is passed:

```
if (isPastCuePoint(sound 1, "Chorus End")=3) then
  puppetSound 0
end if
```

The following example displays information in cast member “field 2” about the music playing in sound channel 1. If the music is not yet past cue point “climax”, the text of “field 2” is “This is the beginning of the piece.” Otherwise, the text reads “This is the end of the piece.”

```
if not sound(1).isPastCuePoint("climax") then
    member("field 2").text = "This is the beginning of the piece."
else
    member("field 2").text = "This is the end of the piece."
end if
```

## isVRMovie

### Syntax

```
member(whichCastMember).isVRMovie
isVRMovie of member whichCastMember
sprite(whichSprite).isVRMovie
isVRMovie of sprite whichSprite
```

### Description

QuickTime cast member and sprite property; indicates whether a cast member or sprite is a QuickTime VR movie that has not yet been downloaded (TRUE), or whether the cast member or sprite isn't a QuickTime VR movie (FALSE).

Testing for this property in anything other than an asset whose type is #quickTimeMedia produces an error message.

This property can be tested but not set.

### Example

The following handler checks to see if the member of a sprite is a QuickTime movie. If it is, the handler further checks to see if it is a QTVR movie. An alert is posted in any case.

```
on checkForVR theSprite
    if sprite(theSprite).member.type = #quickTimeMedia then
        if sprite(theSprite).isVRMovie then
            alert "This is a QTVR asset."
        else
            alert "This is not a QTVR asset."
        end if
    else
        alert "This is not a QuickTime asset."
    end if
end
```

## item...of

### Syntax

```
textMemberExpression.item[whichItem]
item whichItem of fieldOrStringVariable
textMemberExpression.item[firstItem..lastItem]
item firstItem to lastItem of fieldOrStringVariable
```

### Description

Keyword; specifies an item or range of items in a chunk expression. An item in this case is any sequence of characters delimited by the current delimiter as determined by the itemDelimiter property.

The terms *whichItem*, *firstItem*, and *lastItem* must be integers or integer expressions that refer to the position of items in the chunk.

Chunk expressions refer to any character, word, item, or line in any source of strings. Sources of strings include field and text cast members and variables that hold strings.

When the number that specifies the last item is greater than the item's position in the chunk expression, the actual last item is specified instead.

### Examples

This statement looks for the third item in the chunk expression that consists of names of colors and then displays the result in the Message window:

```
put "red, yellow, blue green, orange".item[3]
-- "blue green"
```

The result is the entire chunk “blue green” because this is the entire chunk between the commas.

The following statement looks for the third through fifth items in the chunk expression. Because there are only four items in the chunk expression, only the third item is used and fourth items are returned. The result appears in the Message window.

```
put "red, yellow, blue green, orange".item[3..5]"
-- " blue green, orange"
put item 5 of "red, yellow, blue green, orange"
-- ""
```

The following statement inserts the item Desk as the fourth item in the second line of the field cast member All Bids:

```
member("All Bids").line[2].item[4] = "Desk"
```

### See also

`char...of`, `itemDelimiter`, `number (items)`, `word...of`

## itemDelimiter

### Syntax

the `itemDelimiter`

### Description

System property; indicates the special character used to separate items.

You can use the `itemDelimiter` to parse filenames by setting `itemDelimiter` to a backslash (\) in Windows or a colon (:) on the Macintosh. Restore the `itemDelimiter` character to a comma (,) for normal operation.

This function can be tested and set.

### Example

The following handler finds the last component in a Macintosh pathname. The handler first records the current delimiter and then changes the delimiter to a colon (:). When a colon is the delimiter, Lingo can use the `last item of` to determine the last item in the chunk that makes up a Macintosh pathname. Before exiting, the delimiter is reset to its original value.

```
on getLastComponent pathName
    save = the itemDelimiter
    the itemDelimiter = ":"
    f = the last item of pathName
    the itemDelimiter = save
    return f
end
```

## Kerning

### Syntax

`member(whichTextMember).kerning`

### Description

Text cast member property; this property specifies whether the text is automatically kerned when the contents of the text cast member are changed.

When set to `TRUE`, kerning is automatic; when set to `FALSE`, kerning is not done.

This property defaults to `TRUE`.

### See also

`kerningThreshold`

## kerningThreshold

### Syntax

`member(whichTextMember).kerningThreshold`

### Description

Text cast member property; this setting controls the size at which automatic kerning takes place in a text cast member. This has an effect only when the kerning property of the text cast member is set to `TRUE`.

The setting itself is an integer indicating the font point size at which kerning takes place.

This property defaults to 14 points.

### See also

`kerning`

## key()

### Syntax

`the key`

### Description

Function; indicates the last key that was pressed. This value is the American National Standards Institute (ANSI) value assigned to the key, not the numerical value.

You can use the `key` in handlers that perform certain actions when the user presses specific keys as shortcuts and other forms of interactivity. When used in a primary event handler, the actions you specify are the first to be executed.

**Note:** The value of the `key` isn't updated if the user presses a key while Lingo is in a repeat loop.

Use the sample movie *Keyboard Lingo* to test which characters correspond to different keys on different keyboards.

### Examples

The following statements cause the movie to return to the main menu marker when the user presses the Return key. Because the `keyDownScript` property is set to `checkKey`, the `on prepareMovie` handler makes the `on checkKey` handler the first event handler executed when a key is pressed. The `on checkKey` handler checks whether the Return key is pressed and if it is, navigates to the main menu marker.

```
on prepareMovie
    the keyDownScript = "checkKey"
end prepareMovie
on checkKey
    if the key = RETURN then go to frame "Main Menu"
end
```

This `on keyDown` handler checks whether the last key pressed is the Enter key and if it is, calls the `on addNumbers` handler:

```
on keyDown
    if the key = RETURN then addNumbers
end keyDown
```

### See also

`commandDown`, `controlDown`, `keyCode()`, `optionDown`

## keyboardFocusSprite

### Syntax

```
set the keyboardFocusSprite = textSpriteNum
```

### Description

System property; lets the user set the focus for keyboard input (without controlling the cursor's insertion point) on a particular text sprite currently on the screen. This is the equivalent to using the Tab key when the `AutoTab` property of the member is selected.

Setting `keyboardFocusSprite` to -1 returns keyboard focus control to the score, and setting it to 0 disables keyboard entry into any editable sprite.

### See also

`autoTab`, `editable`

# keyCode()

## Syntax

the keyCode

## Description

Function; gives the numerical code for the last key pressed. This keyboard code is the key's numerical value, not the American National Standards Institute (ANSI) value.

**Note:** When a movie plays back as an applet, this function returns the values of only function and arrow keys.

You can use the `keyCode` function to detect when the user has pressed an arrow or function key, which cannot be specified by the `key` function.

Use the sample movie Keyboard Lingo to test which characters correspond to different keys on different keyboards.

This function can be tested but not set.

## Examples

This handler uses the Message window to display the appropriate key code each time a key is pressed:

```
on enterFrame
    the keydownScript = "put the keyCode"
end
```

This statement checks whether the up arrow (whose key code is 126) was pressed and if it was, goes to the previous marker:

```
if the keyCode = 126 then go to marker(-1)
```

This handler checks whether one of the arrow keys was pressed and if one was, responds accordingly:

```
on keyDown
    case (the keyCode) of
        123: TurnLeft
        126: GoForward
        125: BackUp
        124: TurnRight
    end case
end
```

## See also

`commandDown`, `controlDown`, `key()`, `optionDown`



## on keyDown

### Syntax

```
on keyDown
    statement(s)
end
```

### Description

System message and event handler; contains statements that run when a key is pressed.

When a key is pressed, Director searches these locations, in order, for an `on keyDown` handler: primary event handler, editable field sprite script, field cast member script, frame script, and movie script. For sprites and cast members, `on keyDown` handlers work only for editable text and field members. A `keyDown` event on a different type of cast member, such as a bitmap, has no effect. (If pressing a key should have the same response throughout the movie, set `keyDownScript`.)

Director stops searching when it reaches the first location that has an `on keyDown` handler, unless the handler includes the `pass` command to explicitly pass the `keyDown` message on to the next location.

The `on keyDown` event handler is a good place to put Lingo that implements keyboard shortcuts or other interface features that you want to occur when the user presses keys.

The Director player for Java responds to `keyDown` messages only if the movie has focus in the browser. The user must click in the applet before the applet can receive any keys that the user types.

When the movie plays back as an applet, an `on keyDown` handler always traps key presses, even if the handler is empty. If the user is typing in an editable field, an `on keyDown` handler attached to the field must include the `pass` command for the key to appear in the field.

Where you place an `on keyDown` handler can affect when it runs.

- To apply the handler to a specific editable field sprite, put the handler in a sprite script.
- To apply the handler to an editable field cast member in general, put the handler in a cast member script.
- To apply the handler to an entire frame, put the handler in a frame script.
- To apply the handler throughout the entire movie, put the handler in a movie script.

You can override an `on keyDown` handler by placing an alternative `on keyDown` handler in a location that Lingo checks before it gets to the handler you want to override. For example, you can override an `on keyDown` handler assigned to a cast member by placing an `on keyDown` handler in a sprite script.

### Example

This handler checks whether the Return key was pressed and if it was, sends the playhead to another frame:

```
on keyDown
    if the key = RETURN then go to frame "AddSum"
end keyDown
```

### See also

`charToNum()`, `keyDownScript`, `keyUpScript`, `key()`, `keyCode()`, `keyPressed()`

# keyDownScript

## Syntax

the `keyDownScript`

## Description

System property; specifies the Lingo that is executed when a key is pressed. The Lingo is written as a string, surrounded by quotation marks, and can be a simple statement or a calling script for a handler.

When a key is pressed and the `keyDownScript` property is defined, Lingo executes the instructions specified for the `keyDownScript` property first. Unless the instructions include the `pass` command so that the `keyDown` message can be passed on to other objects in the movie, no other `on keyDown` handlers are executed.

Setting the `keyDownScript` property performs the same function as using the `when keyDown then` command that appeared in earlier versions of Director.

When the instructions you specify for the `keyDownScript` property are no longer appropriate, turn them off by using the statement `set the keyDownScript to EMPTY`.

## Examples

The following statement sets `keyDownScript` to if the key = RETURN then go to the frame + 1. When this statement is in effect, the movie always goes to the next frame whenever the user presses the Return key.

```
the keyDownScript = "if the key = RETURN then go to the frame + 1"
```

The following statement sets `keyDownScript` to the custom handler *myCustomHandler*. A Lingo custom handler must be enclosed in quotation marks when used with the `keyDownScript` property.

```
the keyDownScript = "myCustomHandler"
```

## See also

`on keyDown`, `keyUpScript`, `mouseDownScript`, `mouseUpScript`

# keyframePlayer (modifier)

## Syntax

```
member(whichCastmember).model(whichModel).\  
keyframePlayer.keyframePlayerModifierProperty
```

## Description

3D modifier; manages the use of motions by models. The motions managed by the `keyframePlayer` modifier animate the entire model at once, unlike Bones player motions, which animate segments of the model called bones.

Motions and the models that use them must be created in a 3D modeling program, exported as W3D files, and then imported into a movie. Motions cannot be applied to model primitives created within Director.

Adding the `keyframePlayer` modifier to a model by using the `addModifier` command allows access to the following `keyframePlayer` modifier properties:

- `playing` indicates whether a model is executing a motion.
- `playlist` is a linear list of property lists containing the playback parameters of the motions that are queued for a model.
- `currentTime` indicates the local time, in milliseconds, of the currently playing or paused motion.
- `playRate` is a number that is multiplied by the `scale` parameter of the `play()` or `queue()` command to determine the playback speed of the motion.
- `playlist.count` returns the number of motions currently queued in the playlist.
- `rootLock` indicates whether the translational component of the motion is used or ignored.
- `currentLoopState` indicates whether the motion plays once or repeats continuously.
- `blendTime` indicates the length of the transition created by the modifier between motions when the modifier's `autoBlend` property is set to `TRUE`.
- `autoBlend` indicates whether the modifier creates a linear transition to the currently playing motion from the motion that preceded it.
- `blendFactor` indicates the degree of blending between motions when the modifier's `autoBlend` property is set to `FALSE`.
- `lockTranslation` indicates whether the model can be displaced from the specified planes.
- `positionReset` indicates whether the model returns to its starting position after the end of a motion or each iteration of a loop.
- `rotationReset` indicates the rotational element of a transition from one motion to the next, or the looping of a single motion.

**Note:** For more detailed information about these properties, see the individual property entries.

The `keyframePlayer` modifier uses the following commands:

- `pause` halts the motion currently being executed by the model.
- `play()` initiates or unpauses the execution of a motion.
- `playNext()` initiates playback of the next motion in the playlist.
- `queue()` adds a motion to the end of the playlist.

The `keyframePlayer` modifier generates the following events, which are used by handlers declared in the `registerForEvent()` and `registerScript()` commands. The call to the declared handler includes three arguments: the event type (either `#animationStarted` or `#animationEnded`), the name of the motion, and the current time of the motion. For detailed information about notification events, see the entry for `registerForEvent()`.

`#animationStarted` is sent when a motion begins playing. If blending is used between motions, the event is sent when the transition begins.

`#animationEnded` is sent when a motion ends. If blending is used between motions, the event is sent when the transition ends.

#### See also

`addModifier`, `modifiers`, `bonesPlayer (modifier)`, `motion`

## keyPressed()

### Syntax

```
the keyPressed  
keyPressed (keyCode)  
keyPressed (asciiCharacterString)
```

### Description

Function; returns the character assigned to the key that was last pressed if no argument is used. The result is in the form of a string. When no key has been pressed, the `keyPressed` is an empty string.

If an argument is used, either a `keyCode` or the ASCII string for the key being pressed may be used. In either of these cases, the return value is `TRUE` if that particular key is being pressed, or `FALSE` if not.

The Director player for Java doesn't support this property. As a result, a movie playing back as an applet has no way to detect which key the user pressed while Lingo is in a repeat loop.

The `keyPressed` property is updated when the user presses keys while Lingo is in a repeat loop. This is an advantage over the `key` function, which doesn't update when Lingo is in a repeat loop.

Use the sample movie Keyboard Lingo to test which characters correspond to different keys on different keyboards.

This property can be tested but not set.

### Examples

The following statement checks whether the user pressed the Enter key in Windows or the Return key on a Macintosh and runs the handler `updateData` if the key was pressed:

```
if the keyPressed = RETURN then updateData
```

This statement uses the `keyCode` for the *a* key to test if it's down and displays the result in the Message window:

```
if keyPressed(0) then put "Key is down"
```

This statement uses the ASCII strings to test if the *a* and *b* keys are down and displays the result in the Message window:

```
if keyPressed("a") and keyPressed ("b") then put "Keys are down"
```

### See also

`keyCode()`, `key()`

## on keyUp

### Syntax

```
on keyUp  
    statement(s)  
end
```

### Description

System message and event handler; contains statements that run when a key is released. The `on keyUp` handler is similar to the `on keyDown` handler, except this event occurs after a character appears if a field or text sprite is editable on the screen.

When a key is released, Lingo searches these locations, in order, for an `on keyUp` handler: primary event handler, editable field sprite script, field cast member script, frame script, and movie script. For sprites and cast members, `on keyUp` handlers work only for editable strings. A `keyUp` event on a different type of cast member, such as a bitmap, has no effect. If releasing a key should always have the same response throughout the movie, set `keyUpScript`.

Lingo stops searching when it reaches the first location that has an `on keyUp` handler, unless the handler includes the `pass` command to explicitly pass the `keyUp` message on to the next location.

The `on keyUp` event handler is a good place to put Lingo that implements keyboard shortcuts or other interface features that you want to occur when the user releases keys.

The Director player for Java responds to `keyUp` messages only if the movie has focus in the browser. The user must click in the applet before the applet can receive any keys that the user types.

When the movie plays back as an applet, an `on keyUp` handler always traps key presses, even if the handler is empty. If the user is typing in an editable field, an `on keyUp` handler attached to the field must include the `pass` command for the key to appear in the field.

Where you place an `on keyUp` handler can affect when it runs, as follows:

- To apply the handler to a specific editable field sprite, put it in a behavior.
- To apply the handler to an editable field cast member in general, put it in a cast member script.
- To apply the handler to an entire frame, put it in a frame script.
- To apply the handler throughout the entire movie, put it in a movie script.

You can override an `on keyUp` handler by placing an alternative `on keyUp` handler in a location that Lingo checks before it gets to the handler you want to override. For example, you can override an `on keyUp` handler assigned to a cast member by placing an `on keyUp` handler in a sprite script.

### Example

This handler checks whether the Return key was released and if it was, sends the playhead to another frame:

```
on keyUp
    if the key = RETURN then go to frame "AddSum"
end keyUp
```

### See also

`on keyDown`, `keyDownScript`, `keyUpScript`

## keyUpScript

### Syntax

the `keyUpScript`

### Description

System property; specifies the Lingo that is executed when a key is released. The Lingo is written as a string, surrounded by quotation marks, and can be a simple statement or a calling script for a handler.

When a key is released and the `keyUpScript` property is defined, Lingo executes the instructions specified for the `keyUpScript` property first. Unless the instructions include the `pass` command so that the `keyUp` message can be passed on to other objects in the movie, no other `on keyUp` handlers are executed.

When the instructions you've specified for the `keyUpScript` property are no longer appropriate, turn them off by using the statement `set the keyUpScript to empty`.

#### Examples

The following statement sets `keyUpScript` to if the key = RETURN then go the frame + 1. When this statement is in effect, the movie always goes to the next frame whenever the user presses the Return key.

```
the keyUpScript = "if the key = RETURN then go to the frame + 1"
```

The following statement sets `keyUpScript` to the custom handler `myCustomHandler`. A Lingo custom handler must be enclosed in quotation marks when used with the `keyUpScript` property.

```
the keyUpScript = "myCustomHandler"
```

#### See also

on keyUp

## label()

#### Syntax

```
label(expression)
```

#### Description

Function; indicates the frame associated with the marker label specified by *expression*. The term *expression* should be a label in the current movie; if it's not, this function returns 0.

#### Examples

This statement sends the playhead to the tenth frame after the frame labeled Start:

```
go to label("Start") + 10
```

This statement assigns the frame number of the fourth item in the label list to the variable `whichFrame`:

```
whichFrame = label(the labelList.line[4])
```

#### See also

go, frameLabel, labelList, marker(), play

## labelList

#### Syntax

```
the labelList
```

#### Description

System property; lists the frame labels in the current movie as a Return-delimited string (not a list) containing one label per line. Labels are listed according to their order in the Score. (Because the entries are Return-delimited, the end of the string is an empty line after the last Return. Be sure to remove this empty line if necessary.)

#### Examples

This statement makes a list of frame labels in the content of the field cast member Key Frames:

```
member("Key Frames").text = the labelList
```

This handler determines the label that starts the current scene:

```
on findLastLabel
  aa = label(0)
  repeat with i = 1 to (the labelList.line.count - 1)
    if aa = label(the labelList.line[i]) then
      return the labelList.line[i]
    end if
  end repeat
end
```

**See also**

frameLabel, label(), marker()

## last()

**Syntax**

the last chunk of ( *chunkExpression* )  
the last *chunk* in (*chunkExpression*)

**Description**

Function; identifies the last chunk specified by *chunk* in the chunk expression specified by *chunkExpression*.

Chunk expressions refer to any character, word, item, or line in a container of character. Supported containers are field cast members, variables that hold strings, and specified characters, words, items, lines, and ranges within containers.

**Examples**

This statement identifies the last word of the string “Macromedia, the multimedia company” and displays the result in the Message window:

```
put the last word of "Macromedia, the multimedia company"
```

The result is the word *company*.

This statement identifies the last character of the string “Macromedia, the multimedia company” and displays the result in the Message window:

```
put last char("Macromedia, the multimedia company")
```

The result is the letter *y*.

**See also**

char...of, word...of

## lastChannel

**Syntax**

the lastChannel

**Description**

Movie property; the number of the last channel in the movie, as entered in the Movie Properties dialog box.

This property can be tested but not set.

To see an example of lastChannel used in a completed movie, see the QT and Flash movie in the Learning/Lingo Examples folder inside the Director application folder.

**Example**

This statement displays the number of the last channel of the movie in the Message window:

```
put the lastChannel
```

## lastClick()

**Syntax**

```
the lastClick
```

**Description**

Function; returns the time in ticks (1 tick = 1/60 of a second) since the mouse button was last pressed.

This function can be tested but not set.

**Example**

This statement checks whether 10 seconds have passed since the last mouse click and, if so, sends the playhead to the marker No Click:

```
if the lastClick > 10 * 60 then go to "No Click"
```

**See also**

```
lastEvent(), lastKey, lastRoll, startTimer
```

## lastError

**Syntax**

```
sprite(whichSprite).lastError  
member(whichCastmember).lastError
```

**Description**

RealMedia sprite or cast member property; allows you to get the last error symbol returned by RealPlayer as a Lingo symbol. The error symbols returned by RealPlayer are strings of simple English and provide a starting point for the troubleshooting process. This property is dynamic during playback and can be tested but not set.

The value #PNR\_OK indicates that everything is functioning properly.

**Examples**

The following examples show that the last error returned by RealPlayer for the sprite 2 and the cast member Real was #PNR\_OUTOFMEMORY:

```
put sprite(2).lastError  
-- #PNR_OUTOFMEMORY  
  
put member("Real").lastError  
-- #PNR_OUTOFMEMORY
```

## lastEvent()

**Syntax**

```
the lastEvent
```

**Description**

Function; returns the time in ticks (1 tick = 1/60 of a second) since the last mouse click, rollover, or key press occurred.



**Example**

This statement checks whether 10 seconds have passed since the last mouse click, rollover, or key press and, if so, sends the playhead to the marker Help:

```
if the lastEvent > 10 * 60 then go to "Help"
```

**See also**

lastClick(), lastKey, lastRoll, startTimer

## lastFrame

**Syntax**

the lastFrame

**Description**

Movie property; displays the number of the last frame in the movie.

This property can be tested but not set.

**Example**

This statement displays the number of the last frame of the movie in the Message window:

```
put the lastFrame
```

## lastKey

**Syntax**

the lastKey

**Description**

System property; gives the time in ticks (1 tick = 1/60 of a second) since the last key was pressed.

**Example**

This statement checks whether 10 seconds have passed since the last key was pressed and, if so, sends the playhead to the marker No Key:

```
if the lastKey > 10 * 60 then go to "No Key"
```

**See also**

lastClick(), lastEvent(), lastRoll, startTimer

## lastRoll

**Syntax**

the lastRoll

**Description**

System property; gives the time in ticks (1 tick = 1/60 of a second) since the mouse was last moved.

**Example**

This statement checks whether 45 seconds have passed since the mouse was last moved and, if so, sends the playhead to the marker Attract Loop:

```
if the lastRoll > 45 * 60 then go to "Attract Loop"
```

**See also**

lastClick(), lastEvent(), lastKey, startTimer

## left

### Syntax

`sprite(whichSprite).left`  
the left of sprite *whichSprite*

### Description

Sprite property; identifies the left horizontal coordinate of the bounding rectangle of the sprite specified by *whichSprite*.

Sprite coordinates are measured in pixels, starting with (0,0) at the upper left corner of the Stage.

When a movie plays back as an applet, this property's value is relative to the left edge of the applet.

This property can be tested and set.

### Examples

The following statement determines whether the sprite's left edge is to the left of the Stage's left edge. If the sprite's left edge is to the Stage's left edge, the script runs the handler `offLeftEdge`:

```
if sprite(3).left < 0 then offLeftEdge
```

This statement measures the left horizontal coordinate of the sprite numbered (i + 1) and assigns the value to the variable named `vLowest`:

```
set vLowest = sprite (i + 1).left
```

### See also

`bottom`, `height`, `locH`, `locV`, `right`, `top`, `width`

## left (3D)

### Syntax

`member(whichCastmember).modelResource(whichModelResource).left`

### Description

3D #box model resource property; indicates whether the side of the box intersected by its -X axis is sealed (TRUE) or open (FALSE).

The default value for this property is TRUE.

### Example

This statement sets the `left` property of the model resource named `Crate` to FALSE, meaning the left side of this box will be open:

```
member("3D World").modelResource("crate").left = FALSE
```

### See also

`back`, `front`, `bottom (3D)`, `top (3D)`, `right (3D)`

## leftIndent

### Syntax

*chunkExpression*.leftIndent

### Description

Text cast member property; contains the number of pixels the left margin of *chunkExpression* is offset from the left side of the text cast member.

The value is an integer greater than or equal to 0.

This property can be tested and set.

### Example

This line indents the first line of text cast member “theStory” by ten pixels:

```
member("theStory").line[1].leftIndent = 10
```

### See also

firstIndent, rightIndent

## length()

### Syntax

*string*.length  
length(*string*)

### Description

Function; returns the number of characters in the string specified by *string*, including spaces and control characters such as TAB and RETURN.

### Examples

This statement displays the number of characters in the string “Macro”&“media”:

```
put ("Macro" & "media").length  
-- 10
```

This statement checks whether the content of the field cast member Filename has more than 31 characters and if it does, displays an alert:

```
if member("Filename").text.length > 31 then  
    alert "That filename is too long."  
end if
```

### See also

chars(), offset() (string function)

## length (3D)

### Syntax

member(*whichCastmember*).modelResource(*whichModelResource*).length  
*vectorReference*.length

### Description

3D #box model resource, #plane model resource, and vector property; indicates the length in world units of the box or plane.

The length of a box is measured along its Z axis. The default length of a box is 50.

The length of a plane is measured along its Y axis. The default length of a plane is 1.

The length of a vector is its distance in world units from `vector(0, 0, 0)`. This is the same as the magnitude of the vector.

#### Example

This statement sets the variable `myBoxLength` to the length of the model resource named `GiftBox`.

```
myBoxLength = member("3D World").modelResource("GiftBox").length
```

#### See also

`height (3D)`, `width (3D)`, `magnitude`

## lengthVertices

#### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\lengthVertices
```

#### Description

3D `#box` and `#plane` model resource property; indicates the number of mesh vertices along the length of the box or plane. Increasing this value increases the number of faces, and therefore the fineness, of the mesh.

The length of a box is measured along its Z axis. The length of a plane is measured along its Y axis.

Set the `renderStyle` property of a model's shader to `#wire` to see the faces of the mesh of the model's resource. Set the `renderStyle` property to `#point` to see just the vertices of the mesh.

The value of this property must be greater than or equal to 2. The default value is 4.

#### Example

The following statement sets the `lengthVertices` property of the model resource named `Tower` to 10. Nine triangles will be used to define the geometry of the model resource along its Y axis; therefore, there will be ten vertices.

```
member("3D World").modelResource("Tower").lengthVertices = 10
```

#### See also

`length (3D)`

## level

#### Syntax

```
member(whichCastmember).model(whichModel).lod.level
```

#### Description

3D `lod` modifier property; indicates the amount of detail removed by the modifier when its `auto` property is set to `FALSE`. The range of this property is 0.0 to 100.00.

When the modifier's `auto` property is set to `TRUE`, the value of the `level` property is dynamically updated, but cannot be set.

The `#lod` modifier can only be added to models created outside of Director in 3D modeling programs. The value of the `type` property of the model resources used by these models is `#fromFile`. The modifier cannot be added to primitives created within Director.

### Example

The following statement sets the `level` property of the `lod` modifier of the model `Spaceship` to 50. If the `lod` modifier's `auto` property is set to `FALSE`, `Spaceship` will be drawn at a medium level of detail. If the `lod` modifier's `auto` property is set to `TRUE`, this code will have no effect.

```
member("3D World").model("Spaceship").lod.level = 50
```

### See also

`lod` (modifier), `auto`, `bias`

## lifetime

### Syntax

```
member(whichCastmember).modelResource(modelResource).lifetime
```

### Description

3D #particle model resource property; for all particles in a particle system, this property indicates the number of milliseconds from the creation of a particle to the end of its existence.

The default value of this property is 10,000.

### Example

In this example, `ThermoSystem` is a model resource of the type `#particle`. This statement sets the `lifetime` property of `ThermoSystem` to 90.0 This means each particle of `ThermoSystem` will exist for 90 milliseconds.

```
member(8,2).modelResource("ThermoSystem").lifetime = 90.0
```

### See also

`emitter`

## light

### Syntax

```
member(whichCastmember).light(whichLight)  
member(whichCastmember).light[index]  
member(whichCastmember).light(whichLight).whichLightProperty  
member(whichCastmember).light[index].whichLightProperty
```

### Description

3D element; an object at a vector position from which light emanates.

For a complete list of light properties and commands, see Chapter 2, “3D Lingo by Feature,” on page 31.

### Example

This example shows the two ways of referring to a light. The first line uses a string in parentheses and the second line uses the a number in brackets. The string is the light's name and the number is the position of the light in the cast member's list of lights.

```
thisLight = member("3D World").light("spot01")  
thisLight = member("3D World").light[2]
```

### See also

`newLight`, `deleteLight`

## line...of

### Syntax

```
textMemberExpression.line[whichLine]  
line whichLine of fieldOrStringVariable  
textMemberExpression.line[firstLine..lastLine]  
line firstLine to lastLine of fieldOrStringVariable
```

### Description

Keyword; specifies a line or a range of lines in a chunk expression. A line chunk is any sequence of characters delimited by carriage returns, not by line breaks caused by text wrapping.

The expressions *whichLine*, *firstLine*, and *lastLine* must be integers that specify a line in the chunk.

Chunk expressions refer to any character, word, item, or line in any source of characters. Sources of characters include field cast members and variables that hold strings.

### Examples

This statement assigns the first four lines of the variable `Action` to the field cast member `To Do`:

```
member("To Do").text = Action.line[1..4]
```

This statement inserts the word `and` after the second word of the third line of the string assigned to the variable `Notes`:

```
put "and" after Notes.line[3].word[2]
```

### See also

`char...of`, `item...of`, `word...of`, `number (words)`

## lineColor

### Syntax

```
member(whichCastmember).model(whichModel).inker.lineColor  
member(whichCastmember).model(whichModel).toon.lineColor
```

### Description

3D `toon` and `inker` modifier property; indicates the color of the lines drawn on the model by the modifier. For this property to have an effect, either the modifier's `creases`, `silhouettes`, or `boundary` property must be set to `TRUE`.

The default value for this property is `rgb(0, 0, 0)`.

### Example

This statement sets the color of all lines drawn by the `toon` modifier on the model named `Teapot` to `rgb(255, 0, 0)`, which is red:

```
member("shapes").model("Teapot").toon.lineColor = rgb(255, 0, 0)
```

### See also

`creases`, `silhouettes`, `boundary`, `lineOffset`

## lineCount

### Syntax

`member(whichCastMember).lineCount`  
the lineCount of member *whichCastMember*

### Description

Cast member property; indicates the number of lines that appear in the field cast member on the Stage according to the way the string wraps, not the number of carriage returns in the string.

### Example

This statement determines how many lines the field cast member Today's News has when it appears on the Stage and assigns the value to the variable `numberOfLines`:

```
numberOfLines = member("Today's News").lineCount
```

## lineDirection

### Syntax

`member(whichCastMember).lineDirection`

### Description

Shape member property; this property contains a 0 or 1 indicating the slope of the line drawn.

If the line is inclined from left to right, the property is set to 1; and if it is declined from left to right, the property is set to 0.

This property can be tested and set.

### Example

This handler toggles the slope of the line in cast member "theLine", producing a see-saw effect:

```
on seeSaw
  member("theLine").lineDirection = \
    not member("theLine").lineDirection
end
```

## lineHeight() (function)

### Syntax

`member(whichCastMember).lineHeight(lineNumber)`  
`lineHeight(member whichCastMember, lineNumber)`

### Description

Function; returns the height, in pixels, of a specific line in the specified field cast member.

### Example

This statement determines the height, in pixels, of the first line in the field cast member Today's News and assigns the result to the variable `headline`:

```
headline = member("Today's News").lineHeight(1)
```

## lineHeight (cast member property)

### Syntax

`member(whichCastMember).lineHeight`  
the lineHeight of member *whichCastMember*

### Description

Cast member property; determines the line spacing used to display the specified field cast member. The parameter *whichCastMember* can be either a cast member name or number.

Setting the `lineHeight` member property temporarily overrides the system's setting until the movie closes. To use the desired line spacing throughout a movie, set the `lineHeight` member property in an `on prepareMovie` handler.

This property can be tested and set.

### Example

This statement sets the variable `oldHeight` to the current `lineHeight` setting for the field cast member Rokujo Speaks:

```
oldHeight = member("Rokujo Speaks").lineHeight
```

### See also

`text`, `alignment`, `font`, `fontSize`, `fontStyle`

## lineOffset

### Syntax

`member(whichCastmember).model(whichModel).toon.lineOffset`  
`member(whichCastmember).model(whichModel).inker.lineOffset`

### Description

3D toon and inker modifier property; indicates the apparent distance from the model's surface at which lines are drawn by the modifier. For this property to have an effect, the modifier's `useLineOffset` property must be set to `TRUE`, and one or more of its `creases`, `silhouettes`, or `boundary` properties must also be set to `TRUE`.

This range of this property is -100.00 to +100.00. Its default setting is -2.0.

### Example

The following statement sets the `lineOffset` property of the `toon` modifier for the model named Teapot to 10. The lines drawn by the toon modifier on the surface of the model will stand out more than they would at the default setting of -2.

```
member("shapes").model("Teapot").toon.lineOffset = 10
```

### See also

`creases`, `silhouettes`, `boundary`, `useLineOffset`, `lineColor`



## linePosToLocV()

### Syntax

```
member(whichCastMember).linePosToLocV(lineNumber )  
linePosToLocV(member whichCastMember, lineNumber)
```

### Description

Function; returns a specific line's distance, in pixels, from the top edge of the field cast member.

### Example

This statement measures the distance, in pixels, from the second line of the field cast member Today's News to the top of the field cast member and assigns the result to the variable

startOfString:

```
startOfString = member("Today's News").linePosToLocV(2)
```

## lineSize

### Syntax

```
member(whichCastMember).lineSize  
the lineSize of member whichCastMember  
sprite whichSprite.lineSize  
the lineSize of sprite whichSprite
```

### Description

Shape cast member property; determines the thickness, in pixels, of the border of the specified shape cast member displayed on the Stage. For nonrectangular shapes, the border is the edge of the shape, not its bounding rectangle.

The `lineSize` setting of the sprite takes precedence over the `lineSize` setting of the member. If Lingo changes the member's `lineSize` setting while a sprite is on the Stage, the sprite's `lineSize` setting remains in effect until the sprite is finished.

For the value set by Lingo to last beyond the current sprite, the sprite must be a puppet.

This property can be tested and set.

### Examples

This statement sets the thickness of the shape cast member Answer Box to 5 pixels:

```
member("Answer Box").lineSize = 5
```

This statement displays the thickness of the border of sprite 4:

```
thickness = sprite(4).lineSize
```

This statement sets the thickness of the border of sprite 4 to 3 pixels:

```
sprite(4).lineSize = 3
```

## linkAs()

### Syntax

```
castMember.linkAs()
```

### Description

Script cast member function; opens a save dialog box, allowing you to save the contents of the script to an external file. The script cast member is then linked to that file.

Linked scripts are imported into the movie when you save it as a projector, Shockwave movie, or Java movie. This differs from other linked media, which remains external to the movie unless you explicitly import it.

### Example

These statements, typed in the Message window, opens a Save dialog box to save the script Random Motion as an external file:

```
member("Random Motion").linkAs()  
importFileInto, linked
```

## linked

### Syntax

```
member(whichMember).linked  
the linked of member whichMember
```

### Description

Cast member property; controls whether a script, Flash movie, or animated GIF file is stored in an external file (TRUE, default), or inside the Director cast (FALSE). When the data is stored externally, the cast member's `pathName` property must point to the location where the movie file can be found.

This property can be tested and set for script, Flash, and GIF members. It may be tested for all member types.

### Example

This statement converts Flash cast member “Homebodies” from a linked member to an internally stored member.

```
member("homeBodies").linked = 0
```

### See also

`fileName` (cast member property), `pathName` (cast member property)

## list()

### Syntax

```
list(value1, value2, value3...)
```

### Description

Function and data type; defines a linear list made up of the values specified by *value1*, *value2*, *value3*.... This is an alternative to using square brackets ([ ]) to create a list.

The maximum length of a single line of executable Lingo is 256 characters. You can't create a very large list using this command. If you have a large amount of data that you want to put in a list, enclose the data in square brackets and put the data into a field. You can then assign the field to a variable. The variable's content is a list of the data.

#### Example

This statement sets the variable named `designers` equal to a linear list that contains the names Gee, Kayne, and Ohashi:

```
designers = list("Gee", "Kayne", "Ohashi")
```

The result is the list ["Gee", "Kayne", "Ohashi"].

#### See also

`integer()`, `integerP()`, `value()`

## listP()

#### Syntax

```
listP(item)
```

#### Description

Function; indicates whether the item specified by *item* is a list, rectangle, or point (1 or TRUE) or not (0 or FALSE).

#### Example

This statement checks whether the list in the variable `designers` is a list, rectangle, or point, and displays the result in the Message window:

```
put listP(designers)
```

The result is 1, which is the numerical equivalent of TRUE.

#### See also

`ilk()`, `objectP()`

## loaded

#### Syntax

```
member(whichCastMember).loaded  
the loaded of member whichCastMember
```

#### Description

Cast member property; specifies whether the cast member specified by *whichCastMember* is loaded into memory (TRUE) or not (FALSE).

Different cast member types have slightly different behaviors for loading:

- Shape and script cast members are always loaded into memory.
- Movie cast members are never unloaded.
- Digital video cast members can be preloaded and unloaded independent of whether they are being used. (A digital video cast member plays faster from memory than from disk.)

This property can be tested but not set.

### Example

This statement checks whether cast member Demo Movie is loaded in memory and if it isn't, goes to an alternative movie:

```
if member("Demo Movie").loaded = FALSE then go to movie("Waiting")"
```

### See also

`preLoad (command)`, `ramNeeded()`, `size`, `unLoad`

## loadFile()

### Syntax

```
member(whichCastmember).loadFile(fileName {, overwrite, \
    generateUniqueNames})
```

### Description

3D cast member command; imports the assets of the W3D file, *fileName*, into the cast member.

The optional *overwrite* parameter indicates whether the assets of the W3D file replace the assets of the cast member (TRUE) or are added to the assets of the cast member (FALSE). The default value of *overwrite* is TRUE.

If the optional *generateUniqueNames* parameter is set to TRUE, any element in the W3D file with the same name as a corresponding element in the cast member is renamed. If *generateUniqueNames* is FALSE, elements in the cast member are overwritten by corresponding elements in the W3D file with the same name. The default value of *generateUniqueNames* is TRUE.

The cast member's *state* property must be either -1 (error) or 4 (loaded) before the `loadFile` command is used.

### Examples

The following statement imports the contents of the file named `Truck.W3d` into the cast member named `Roadway`. The contents of `Truck.W3d` will be added to the contents of `Roadway`. If any imported objects have the same names as objects already in `Roadway`, Director will create new names for them.

```
member("Roadway").loadFile("Truck.W3d", FALSE, TRUE)
```

The following statement imports the contents of the file named `Chevy.W3d` into the cast member named `Roadway`. `Chevy.W3d` is in a folder named `Models` one level down from the movie. The contents of `Roadway` will be replaced by the contents of `Chevy.W3d`. The third parameter is irrelevant because the value of the second parameter is TRUE.

```
member("Roadway").loadFile(the moviePath & "Models\Chevy.W3d", \
    TRUE, TRUE)
```

### See also

`state (3D)`

## loc

### Syntax

`sprite whichSprite.loc`  
the loc of sprite *whichSprite*

### Description

Sprite property; determines the Stage coordinates of the specified sprite's registration point. The value is given as a point.

This property can be tested and set.

To see an example of `loc` used in a completed movie, see the Imaging movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This statement checks the Stage coordinates of sprite 6. The result is the point (50, 100):

```
put sprite(6).loc  
-- point(50, 100)
```

### See also

`bottom`, `height`, `left`, `locH`, `locV`, `right`, `top`, `width`

## loc (backdrop and overlay)

### Syntax

```
sprite(whichSprite).camera{( index )}.backdrop[ index ].loc  
member(whichCastmember).camera(whichCamera).backdrop[ index ].loc  
sprite(whichSprite).camera{( index )}.overlay[ index ].loc  
member(whichCastmember).camera(whichCamera).overlay[ index ].loc
```

### Description

3D backdrop and overlay property; indicates the 2D location of the backdrop or overlay, as measured from the upper left corner of the sprite.

This property is initially set as a parameter of the `addBackdrop`, `addOverlay`, `insertBackdrop`, or `insertOverlay` command which creates the backdrop or overlay.

### Example

This statement positions the first backdrop of the camera of sprite 2:

```
sprite(2).camera.backdrop[1].loc = point(120, 120)
```

### See also

`bevelDepth`, `overlay`, `regPoint` (3D)

# locH

## Syntax

`sprite(whichSprite).locH`  
the locH of sprite *whichSprite*

## Description

Sprite property; indicates the horizontal position of the specified sprite's registration point. Sprite coordinates are relative to the upper left corner of the Stage.

This property can be tested and set. To make the value last beyond the current sprite, make the sprite a puppet.

## Examples

This statement checks whether the horizontal position of sprite 9's registration point is to the right of the right edge of the Stage and moves the sprite to the left edge of the Stage if it is:

```
if sprite(9).locH > (the stageRight - the stageLeft) then
    sprite(9).locH = 0
end if
```

This statement puts sprite 15 at the same horizontal location as the mouse click:

```
sprite(15).locH = the mouseH
```

## See also

bottom, height, left, loc, locV, point(), right, top, updateStage, width

# locToCharPos()

## Syntax

`member(whichCastMember). locToCharPos(location )`  
`locToCharPos(member whichCastMember, location)`

## Description

Function; returns a number that identifies which character in the specified field cast member is closest to the point within the field specified by *location*. The value for *location* is a point relative to the upper left corner of the field cast member.

The value 1 corresponds to the first character in the string, the value 2 corresponds to the second character in the string, and so on.

## Examples

The following statement determines which character is closest to the point 100 pixels to the right and 100 pixels below the upper left corner of the field cast member Today's News. The statement then assigns the result to the variable PageDesign.

```
pageDesign = member("Today's News").locToCharPos(point(100, 100))
```

This handler tells which character is under the pointer when the user clicks the mouse over the field sprite Information:

```
on mouseDown
    put member("Information").locToCharPos(the clickLoc - \
        (sprite(the clickOn).loc))
end
```

## lockTranslation

### Syntax

```
member(whichCastmember).model(whichModel).bonesPlayer.\
    lockTranslation
member(whichCastmember).model(whichModel).keyframePlayer.\
    lockTranslation
```

### Description

3D `#bonesPlayer` and `#keyframePlayer` modifier property; prevents displacement from the specified plane(s) except by the absolute translation of the motion data. Any additional translation introduced either manually or through cumulative error is removed. The possible values of `#none`, `#x`, `#y`, `#z`, `#xy`, `#yz`, `#xz`, and `#all` control which of the three translational components are controlled for each frame. When a lock on an axis is turned on, the current displacement along that axis is stored and used thereafter as the fixed displacement to which the animation is relative. This displacement can be reset by deactivating that axis lock, moving the object, and reactivating that axis lock.

In other words, it defines the axis of translation to ignore when playing back a motion. To keep a model locked to a ground plane with the top pointing along the *z*-axis, set `lockTranslation` to `#z`. The default value for this property is `#none`.

### Example

This statement sets the `lockTranslation` property of the model named Walker to `#z`.

```
member("ParkScene").model("Walker").bonesPlayer.\
    lockTranslation = #z
```

### See also

`immovable`

## locV

### Syntax

```
sprite(whichSprite).locV
the locV of sprite whichSprite
```

### Description

Sprite property; indicates the vertical position of the specified sprite's registration point. Sprite coordinates are relative to the upper left corner of the Stage.

This property can be tested and set. To make the value last beyond the current sprite, make the sprite a puppet.

### Example

This statement checks whether the vertical position of sprite 9's registration point is below the bottom of the Stage and moves the sprite to the top of the Stage if it is:

```
if sprite(9).locV > (the stageBottom - the stageTop) then
    sprite(9).locV = 0
end if
```

### Example

This statement puts sprite 15 at the same vertical location as the mouse click:

```
sprite(15).locV = the mouseV
```

### See also

bottom, height, left, loc, locH, point(), right, top, updateStage, width

## locVToLinePos()

### Syntax

```
member(whichCastMember). locVToLinePos(locV )
locVToLinePos(member whichCastMember, locV)
```

### Description

Function; returns the number of the line of characters that appears at the vertical position specified by *locV*. The *locV* value is the number of pixels from the top of the field cast member, not the part of the field cast member that currently appears on the Stage.

### Example

This statement determines which line of characters appears 150 pixels from the top of the field cast member Today's News and assigns the result to the variable *pageBreak*:

```
pageBreak = member("Today's News").locVToLinePos(150)
```

## locZ of sprite

### Syntax

```
sprite(whichSprite).locZ
```

### Description

Sprite property; specifies the dynamic Z-order of a sprite, to control layering without having to manipulate sprite channels and properties.

This property can be tested and set.

This property can have an integer value from negative 2 billion to positive 2 billion. Larger numbers cause the sprite to appear in front of sprites with smaller numbers. If two sprites have the same *locZ* value, the channel number then takes precedence for deciding the final display order of those two sprites. This means sprites in lower numbered channels appear behind sprites in higher numbered channels even when the *locZ* values are equal.

By default, each sprite has a *locZ* value equal to its own channel number.

Layer-dependent operations such as hit detection and mouse events obey sprites' *locZ* values, so changing a sprite's *locZ* value can make the sprite partially or completely obscured by other sprites and the user may be unable to click on the sprite.



Other Director functions do not follow the `locZ` ordering of sprites. Generated events still begin with channel 1 and increase consecutively from there, regardless of the sprite's Z-order.

### Example

This handler uses a global variable called `gHighestSprite` which has been initialized in the `startMovie` handler to the number of sprites used. When the sprite is clicked, its `locZ` is set to `gHighestSprite + 1`, which moves the sprite to the foreground on the stage. Then `gHighestSprite` is incremented by 1 to prepare for the next `mouseUp` call.

```
on mouseUp me
    global gHighestSprite
    sprite(me.spriteNum).locZ = gHighestSprite + 1
    gHighestSprite = gHighestSprite + 1
end
```

### See also

`locH`, `locV`

## lod (modifier)

### Syntax

`member(whichCastmember).model(whichModel).lod.lodModifierProperty`

### Description

3D modifier; dynamically removes detail from models as they move away from the camera.

This modifier can only be added to models created outside of Director in 3D modeling programs. The value of the `type` property of the model resources used by these models is `#fromFile`. All such models use detail reduction whether or not the `lod` modifier is attached. Attaching the modifier allows you to control the properties of detail reduction. The modifier cannot be added to primitives created within Director.

The `lod` modifier data is generated by 3D modeling programs for all models. Setting the `userData` property `"sw3d_no_lod = true"` allows you to specify that the `lod` modifier data and memory be released when streaming is complete.

Be careful when using the `sds` and `lod` modifiers together, because they perform opposite functions (the `sds` modifier adds geometric detail and the `lod` modifier removes geometric detail). Before adding the `sds` modifier, it is recommended that you disable the `lod.auto` modifier property and set the `lod.level` modifier property to maximum resolution, as follows:

```
member("myMember").model("myModel").lod.auto = 0
member("myMember").model("myModel").lod.level = 100
member("myMember").model("myModel").addmodifier(#sds)
```

The `lod` modifier has the following properties:

- `auto` allows the modifier to set the level of detail reduction as the distance between the model and the camera changes. The value of the modifier's `level` property is updated, but setting the `level` property will have no effect when the `auto` property is set to `TRUE`.
- `bias` indicates how aggressively the modifier removes detail from the model when the modifier's `auto` property is set to `TRUE`. The range for this property is from 0.0 (removes all polygons) to 100.0 (removes no polygons). The default setting for this property is 100.0.
- `level` indicates the amount of detail reduction there will be when the modifier's `auto` property is set to `FALSE`. The range of this property is 0.0 to 100.00.

**Note:** For more detailed information about these properties, see the individual property entries.

**See also**

`sds (modifier)`, `auto`, `bias`, `level`, `addModifier`

## log()

**Syntax**

`log(number)`

**Description**

Math function; calculates the natural logarithm of the number specified by *number*, which must be a decimal number greater than 0.

**Example**

This statement assigns the natural logarithm of 10.5 to the variable `Answer`.

```
Answer = log(10.5)
```

**Example**

This statement calculates the natural logarithm of the square root of the value `Number` and then assigns the result to the variable `Answer`:

```
Answer = log(Number.sqrt)
```

## long

**See**

`date()` (`system clock`), `time()`

## loop (cast member property)

**Syntax**

`member(whichCastmember).loop`

**Description**

3D cast member property; indicates whether motions applied to the first model in the cast member repeat continuously (`TRUE`) or play once and stop (`FALSE`).

The default setting for this property is `TRUE`.

**Example**

This statement sets the loop property of the cast member named Walkers to TRUE. Motions being executed by the first model in Walker will repeat continuously.

```
member("Walkers").loop = TRUE
```

**See also**

motion, play() (3D), queue() (3D), animationEnabled

## loop (emitter)

**Syntax**

```
member(whichCastmember).modelResource(whichModelResource).\
    emitter.loop
```

**Description**

3D property; when used with a model resource whose type is #particle, this property allows you to both get and set what happens to particles at the end of their lifetime. A loop value of TRUE causes particles to be reborn at the end of their lifetime at the emitter location defined by the emitter's region property. A value of FALSE causes the particles to die at the end of their lifetime. The default setting for this property is TRUE.

**Example**

In this example, ThermoSystem is a model resource of the type #particle. This statement sets the emitter.loop property of ThermoSystem to 1, which causes the particles of ThermoSystem to be continuously emitted.

```
member("Fires").modelResource("ThermoSystem").emitter.loop = 1
```

**See also**

emitter

## loop (keyword)

**Syntax**

```
loop
```

**Description**

Keyword; refers to the marker. The loop keyword with the go to command is equivalent to the statement go to the marker.

**Example**

This handler loops the movie between the previous marker and the current frame:

```
on exitFrame
    go loop
end exitFrame
```

## loop (cast member property)

### Syntax

`member(whichCastMember).loop`  
the loop of member *whichCastMember*

### Description

Cast member property; determines whether the specified digital video, sound, or Flash movie cast member is set to loop (TRUE) or not (FALSE).

### Example

This statement sets the QuickTime movie cast member Demo to loop:

```
member("Demo").loop = 1
```

## loop (Flash property)

### Syntax

`sprite(whichFlashSprite).loop`  
the loop of sprite *whichFlashSprite*  
`member(whichFlashMember).loop`  
the loop of member *whichFlashMember*

### Description

Flash sprite and member property; controls whether a Flash movie plays in a continuous loop (TRUE) or plays once and then stops (FALSE).

The property can be both tested and set.

### Example

This frame script checks the download status of a linked Flash cast member called NetFlash using the `percentStreamed` property. While NetFlash is downloading, the movie loops in the current frame. When NetFlash finishes downloading, the movie advances to the next frame and the `loop` property of the Flash movie in channel 6 is set to FALSE so that it will continue playing through to the end and then stop (imagine that this sprite has been looping while NetFlash was downloading).

```
on exitFrame
  if member("NetFlash").percentStreamed = 100 then
    sprite(6).loop = FALSE
    go the frame + 1
  end if
  go the frame
end
```

## loopBounds

### Syntax

`sprite(whichQuickTimeSprite).loopBounds`  
the loopBounds of sprite *whichQuickTimeSprite*

### Description

QuickTime sprite property; sets the internal loop points for a QuickTime cast member or sprite. The loop points are specified as a Director list: [*startTime*, *endTime*].

The *startTime* and *endTime* parameters must meet these requirements:

- Both parameters must be integers that specify times in Director ticks.
- The values must range from 0 to the duration of the QuickTime cast member.
- The starting time must be less than the ending time.

If any of these requirements is not met, the QuickTime movie loops through its entire duration.

The `loopBounds` property has no effect if the movie's `loop` property is set to `FALSE`. If the `loop` property is set to `TRUE` while the movie is playing, the movie continues to play. Director uses these rules to decide how to loop the movie:

- If the ending time specified by `loopBounds` is reached, the movie loops back to the starting time.
- If the end of the movie is reached, the movie loops back to the start of the movie.

If the `loop` property is turned off while the movie is playing, the movie continues to play. Director stops when it reaches the end of the movie.

This property can be tested and set. The default setting is `[0,0]`.

#### Example

This sprite script sets the starting and ending times for looping within a QuickTime sprite. Notice that the times are set by specifying seconds, which are then converted to ticks by multiplying by 60.

```
on beginSprite me
    sprite(me.spriteNum).loopBounds = [(16 * 60),(32 * 60)]
end
```

## loopCount

#### Syntax

`sound(channelNum).loopCount`  
the `loopCount` of sound *channelNum*

#### Description

Cast member property; the total number of times the current sound in sound channel *channelNum* is set to loop. The default is 1 for sounds that are simply queued with no internal loop.

You can loop a portion of a sound by passing the parameters `loopStartTime`, `loopEndTime`, and `loopCount` with a `queue()` or `setPlayList()` command. These are the only methods for setting this property.

If `loopCount` is set to 0, the loop will repeat forever. If the sound cast member's `loop` property is set to `TRUE`, the `loopCount` will return 0.

### Example

This handler queues and plays two sounds in sound channel 2. The first sound, cast member `introMusic`, loops five times between 8 seconds and 8.9 seconds. The second sound, cast member `creditsMusic`, loops three times. However, no `#loopStartTime` and `#loopEndTime` are specified, so these values default to the `#startTime` and `#endTime`, respectively.

```
on playMusic
  sound(2).queue([#member:member("introMusic"), #startTime:3000,\
    #loopCount:5,#loopStartTime:8000, #loopEndTime:8900])
  sound(2).queue([#member:member("creditsMusic"), #startTime:3000,\
    #endTime:8000, #loopCount:3])
  sound(2).play()
end
```

### Example

This handler displays an alert indicating how many times the loop in the cast member of sound 2 plays. If no loop has been set in the current sound of sound channel 2, `sound(2).loopCount` returns 1.

```
on showLoopCount
  alert "The current sound's loop plays" && sound(2).loopCount && "times."
end
```

### See also

`breakLoop()`, `setPlaylist()`, `loopEndTime`, `loopsRemaining`, `loopStartTime`, `queue()`

## loopEndTime

### Syntax

`sound(channelNum).loopEndTime`  
the `loopEndTime` of sound `channelNum`

### Description

Sound property; the end time, in milliseconds, of the loop set in the current sound playing in sound channel `channelNum`. Its value is a floating-point number, allowing you to measure and control sound playback to fractions of a millisecond.

This property can only be set when passed as a property in a `queue()` or `setPlaylist()` command.

### Example

This handler plays sound cast member `introMusic` in sound channel 2. Playback loops five times between the 8 seconds point and the 8.9 second point in the sound.

```
on playMusic
  sound(2).play([#member:member("introMusic"), #startTime:3000,\
    #loopCount:5,#loopStartTime:8000, #loopEndTime:8900])
end
```

### Example

This handler causes the text field `TimWords` to read "Help me, I'm stuck!" when the `currentTime` of sound channel 2 is between its `loopStartTime` and `loopEndTime`.

```
on idle
  if sound(2).currentTime > sound(2).loopStartTime and \
    sound(2).currentTime < sound(2).loopEndTime then
    member("TimWords").text = "Help me, I'm stuck!"
  else
    member("TimWords").text = "What's this sticky stuff?"
  end if
end
```

### See also

`breakLoop()`, `getPlaylist()`, `loopCount`, `loopsRemaining`, `loopStartTime`, `queue()`

## loopsRemaining

### Syntax

`sound(channelNum).loopsRemaining`  
the `loopsRemaining` of `sound(channelNum)`

### Description

Read-only property; the number of times left to play a loop in the current sound playing in sound channel `channelNum`. If the sound had no loop specified when it was queued, this property is 0. If this property is tested immediately after a sound starts playing, it returns one less than the number of loops defined with the `#loopCount` property in the `queue()` or `setPlayList()` command.

### See also

`breakLoop()`, `loopCount`, `loopEndTime`, `loopStartTime`, `queue()`

## loopStartTime

### Syntax

`sound(channelNum).loopStartTime`  
the `loopStartTime` of `sound(channelNum)`

### Description

Cast member property; the start time, in milliseconds, of the loop for the current sound being played by *soundObject*. Its value is a floating-point number, allowing you to measure and control sound playback to fractions of a millisecond. The default is the `startTime` of the sound if no loop has been defined.

This property can only be set when passed as a property in a `queue()` or `setPlayList()` command.

### Example

This handler plays sound cast member `introMusic` in sound channel 2. Playback loops five times between two points 8 seconds and 8.9 seconds into the sound.

```
on playMusic
  sound(2).play([#member:member("introMusic"), #startTime:3000,\
    #loopCount:5,#loopStartTime:8000, #loopEndTime:8900])
end
```

**Example**

This handler causes the text field TimWords to read "Help me, I'm stuck!" when the `currentTime` of sound channel 2 is between its `loopStartTime` and `loopEndTime`:

```
on idle
  if sound(2).currentTime > sound(2).loopStartTime and \
    sound(2).currentTime < sound(2).loopEndTime then
    member("TimWords").text = "Help me, I'm stuck!"
  else
    member("TimWords").text = "What's this sticky stuff?"
  end if
end
```

**See also**

`breakLoop()`, `setPlaylist()`, `loopCount`, `loopEndTime`, `loopsRemaining`, `queue()`



## magnitude

### Syntax

*whichVector*.magnitude

### Description

3D property; returns the magnitude of a vector. The value is a floating-point number. The magnitude is the length of a vector and is always greater than or equal to 0.0. (vector (0, 0, 0) equals 0.)

### Example

This statement shows that the magnitude of MyVec1 is 100.0000 and the magnitude of MyVec2 is 141.4214.

```
MyVec1 = vector(100, 0, 0)
put MyVec1.magnitude
-- 100.0000
MyVec2 = vector(100, 100, 0)
put MyVec2.magnitude
-- 141.4214
```

### See also

length (3D), identity()

## makeList()

### Syntax

*parserObject*.makeList()

### Description

Function; returns a property list based on the XML document parsed using parseString() or parseURL().

### Example

This handler parses of an XML document and returns the resulting list:

```
on ConvertToList xmlString
  parserObject = new(xtra "xmlparser")
  errorCode = parserObj.parseString(xmlString)
  errorString = parserObj.getError()
  if voidP(errorString) then
    parsedList = parserObj.makeList()
  else
    alert "Sorry, there was an error" && errorString
    exit
  end if
  return parsedList
end
```

### See also

makeSubList()

## makeSubList()

### Syntax

```
XMLnode.makeSubList()
```

### Description

Function; returns a property list from a child node the same way that `makeList()` returns the root of an XML document in list format.

### Example

Beginning with the following XML:

```
<?xml version="1.0"?>
  <e1>
    <tagName attr1="val1" attr2="val2"/>
    <e2>element 2</e2>
    <e3>element 3</e3>
  </e1>
```

This statement returns a property list made from the contents of the first child of the tag `<e1>`:

```
put gparser.child[ 1 ].child[ 1 ].makeSubList()
-- ["tagName": ["!ATTRIBUTES": ["attr1": "val1", "attr2": "val2"]]]
```

### See also

`makeList()`

## map()

### Syntax

```
map(targetRect, sourceRect, destinationRect)
map(targetPoint, sourceRect, destinationRect)
```

### Description

Function; positions and sizes a rectangle or point based on the relationship of a source rectangle to a target rectangle.

The relationship of the `targetRect` to the `sourceRect` governs the relationship of the result of the function to the `destinationRect`.

### Example

In this behavior, all of the sprites have already been set to draggable. Sprite 2b contains a small bitmap. Sprite 1s is a rectangular shape sprite large enough to easily contain sprite 2b. Sprite 4b is a larger version of the bitmap in sprite 2b. Sprite 3s is a larger version of the shape in sprite 1s. Moving sprite 2b or sprite 1s will cause sprite 4b to move. When you drag sprite 2b, its movements are mirrored by sprite 4b. When you drag sprite 1s, sprite 4b moves in the opposite direction. Resizing sprite 2b or sprite 1s will also produce interesting results.

```
on exitFrame
  sprite(4b).rect = map(sprite(2b).rect, sprite(1s).rect, sprite(3s).rect)
  go the frame
end
```

## map (3D)

### Syntax

```
member(whichCastmember).motion(whichMotion).\  
    map(whichOtherMotion {, boneName})
```

### Description

3D motion command; maps the motion specified by *whichOtherMotion* into the current motion (*whichMotion*), and applies it to the bone specified by *boneName* and all of the children of that bone. This command replaces any motion previously mapped to the specified bone and its children. This command does not change a model's playlist.

The *boneName* parameter defaults to the root bone if not specified.

### Example

The following statement maps the motion named LookUp into the motion named SitDown starting from the bone named Neck. The model will sit down and look up at the same time.

```
member("Restaurant").motion("SitDown").map("LookUp", "Neck")
```

### See also

motion, duration (3D), cloneMotionFromCastmember

## mapMemberToStage()

### Syntax

```
sprite(whichSpriteNumber). mapMemberToStage(whichPointInMember)  
mapMemberToStage(sprite whichSpriteNumber, whichPointInMember)
```

### Description

Function; uses the specified sprite and point to return an equivalent point inside the dimensions of the Stage. This properly accounts for the current transformations to the sprite using quad, or the rectangle if not transformed.

This is useful for determining if a particular area of a cast member has been clicked, even if there have been major transformations to the sprite on the Stage.

If the specified point on the Stage is not within the sprite, a VOID is returned.

### See also

map(), mapStageToMember()

## mapStageToMember()

### Syntax

```
sprite(whichSpriteNumber).mapStageToMember(whichPointOnStage)  
mapStageToMember(sprite whichSpriteNumber, whichPointOnStage)
```

### Description

Function; uses the specified sprite and point to return an equivalent point inside the dimensions of the cast member. This properly accounts for any current transformations to the sprite using quad, or the rectangle if not transformed.

This is useful for determining if a particular area on a cast member has been clicked even if there have been major transformations to the sprite on the Stage.

If the specified point on the Stage is not within the sprite, this function returns VOID.

### See also

map(), mapMemberToStage()

## margin

### Syntax

```
member(whichCastMember).margin  
the margin of member whichCastMember
```

### Description

Field cast member property; determines the size, in pixels, of the margin inside the field box.

### Example

The following statement sets the margin inside the box for the field cast member Today's News to 15 pixels:

```
member("Today's News").margin = 15
```

## marker()

### Syntax

```
marker(integerExpression)  
marker("string")
```

### Description

Function; returns the frame number of markers before or after the current frame. This function is useful for implementing a Next or Previous button or for setting up an animation loop.

The argument *integerExpression* can evaluate to any positive or negative integer or 0. For example:

- marker(2)—Returns the frame number of the second marker after the current frame.
- marker(1)—Returns the frame number of the first marker after the current frame.
- marker(0)—Returns the frame number of the current frame if the current frame is marked, or the frame number of the previous marker if the current frame is not marked.
- marker(-1)—Returns the frame number of the first marker before the marker(0).
- marker(-2)—Returns the frame number of the second marker before the marker(0).

If the argument for `marker` is a string, `marker` returns the frame number of the first frame whose marker label matches the string.

### Examples

The following statement sends the playhead to the beginning of the current frame if the current frame has a marker; otherwise, it sends the playhead to the previous marker.

```
go to marker(0)
```

This statement sets the variable `nextMarker` equal to the next marker in the Score:

```
nextMarker = marker(1)
```

### See also

`go`, `frame()` (function), `frameLabel`, `label()`, `labelList`

## the markerList

### Syntax

the `markerList`

### Description

Global property; contains a Lingo property list of the markers in the Score. The list is of the format:

```
frameNumber: "markerName"
```

This property can be tested but not set.

### Example

This statement displays the list of markers in the Message window:

```
put the markerlist
-- [1: "Opening Credits", 15: "Main Menu", 26: "Closing Credits"]
marker()
```

## mask

### Syntax

```
member(whichQuickTimeMember).mask
the mask of member whichQuickTimeMember
```

### Description

Cast member property; specifies a black-and-white (1-bit) cast member to be used as a mask for media rendered direct to Stage with media appearing in the areas where the mask's pixels are black. The `mask` property lets you benefit from the performance advantages of a direct-to-Stage digital video while playing a QuickTime movie in a nonrectangular area. The `mask` property has no effect on non-direct-to-Stage cast members.

Director always aligns the registration point of the mask cast member with the upper left of the QuickTime movie sprite. Be sure to reset the registration point of a bitmap to the upper left corner, as it defaults to the center. The registration point of the QuickTime member cannot be reset from the upper left corner. The mask cast member can't be moved and is not affected by the center and crop properties of its associated cast member.

For best results, set a QuickTime cast member's mask property before any of its sprites appear on the Stage in the `on beginSprite` event handler. Setting or changing the `mask` property while the cast member is on the Stage can have unpredictable results (for example, the mask may appear as a freeze frame of the digital video at the moment the `mask` property took effect).

Masking is an advanced feature; you may need to experiment to achieve your goal.

This property can be tested and set. To remove a mask, set the `mask` property to 0.

#### Example

This frame script sets a mask for a QuickTime sprite before Director begins to draw the frame:

```
on prepareFrame
    member("Peeping Tom").mask = member("Keyhole")
end
```

#### See also

`invertMask`

## max()

#### Syntax

```
list.max()
max(list)
max(value1, value2, value3, ...)
```

#### Description

Function; returns the highest value in the specified list or the highest of a given series of values.

The `max` function also works with ASCII characters, similar to the way `<` and `>` operators work with strings.

#### Example

The following handler assigns the variable `Winner` the maximum value in the list `Bids`, which consists of `[#Castle:600, #Schmitz:750, #Wang:230]`. The result is then inserted into the content of the field cast member `Congratulations`.

```
on findWinner Bids
    Winner = Bids.max()
    member("Congratulations").text = \
        "You have won, with a bid of $" & Winner & "!"
end
```

## maxInteger

#### Syntax

the `maxInteger`

#### Description

System property; returns the largest whole number that is supported by the system. On most personal computers, this is 2,147,483,647 (2 to the thirty-first power, minus 1).

This property can be useful for initializing boundary variables before a loop or for limit testing.

To use numbers larger than the range of addressable integers, use floating-point numbers instead. They aren't processed as quickly as integers, but they support a greater range of values.

### Example

This statement generates a table, in the Message window, of the maximum decimal value that can be represented by a certain number of binary digits:

```
on showMaxValues
  b = 31
  v = the maxInteger
  repeat while v > 0
    put b && "-" && v
    b = b-1
    v = v/2
  end repeat
end
```

## maxSpeed

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
  emitter.maxSpeed
```

### Description

3D property; when used with a model resource whose type is `#particle`, allows you to get and set the maximum speed at which particles are emitted. Each particle's initial velocity is randomly selected between the emitter's `minSpeed` and `maxSpeed` properties.

The value is a floating-point number and must be greater than 0.0.

### Example

In this example, `ThermoSystem` is a model resource of the type `#particle`. This statement sets the `maxSpeed` property of `ThermoSystem` to 15, which causes the fastest particles of `ThermoSystem` to move fairly quickly. Within a given particle system, the faster a particle moves, the farther it will travel.

```
member("Fires").modelResource("ThermoSystem").emitter.maxSpeed=15
```

### See also

`minSpeed`, `emitter`

## mci

### Syntax

```
mci "string"
```

### Description

Command; for Windows only, passes the strings specified by *string* to the Windows Media Control Interface (MCI) for control of multimedia extensions.

**Note:** Microsoft no longer recommends using the 16-bit MCI interface. Consider using third-party Xtra extensions for this functionality instead.

### Example

The following statement makes the command `play cdaudio from 200 to 600 track 7` play only when the movie plays back in Windows:

```
mci "play cdaudio from 200 to 600 track 7"
```

# me

## Syntax

me

## Description

Special variable; used within parent scripts and behaviors to refer to the current object that is an instance of the parent script or the behavior or a variable that contains the memory address of the object.

The term has no predefined meaning in Lingo. The term `me` is used by convention.

To see an example of `me` used in a completed movie, see the Parent Scripts movie in the Learning/Lingo Examples folder inside the Director application folder.

## Examples

The following statement sets the object `myBird1` to the script named Bird. The `me` keyword accepts the parameter script Bird and is used to return that parameter.

```
myBird1 = new(script "Bird")
```

This is the `on new` handler of the Bird script:

```
on new me
    return me
end
```

The following two sets of handlers make up a parent script. The first set uses `me` to refer to the child object. The second set uses the variable `myAddress` to refer to the child object. In all other respects, the parent scripts are the same.

This is the first set:

```
property myData

on new me, theData
    myData = theData
    return me
end

on stepFrame me
    ProcessData me
end
```

This is the second set:

```
property myData

on new myAddress, theData
    myData = theData
    return myAddress
end

on stepFrame myAddress
    ProcessData myAddress
end
```

## See also

`new()`, `ancestor`



## media

### Syntax

`member(whichCastMember).media`  
the media of member *whichCastMember*

### Description

Cast member property; identifies the specified cast member as a set of numbers. Because setting the `media` member property can use large amounts of memory, this property is best used during authoring only.

You can use the `media` member property to copy the content of one cast member into another cast member by setting the second cast member's `media` value to the `media` value for the first cast member.

For a film loop cast member, the `media` member property specifies a selection of frames and channels in the Score.

To swap media in a projector, it's more efficient to set the `member` sprite property.

### Example

This statement copies the content of the cast member Sunrise into the cast member Dawn by setting the `media` member property value for Dawn to the `media` member property value for Sunrise:

```
member("Dawn").media = member("Sunrise").media
```

### See also

`type (cast member property), media`

## mediaReady

### Syntax

`member(whichCastMember).mediaReady`  
the `mediaReady` of member *whichCastMember*  
`sprite(whichSpriteNumber).mediaReady`  
the `mediaReady` of sprite *whichSpriteNumber*

### Description

Cast member or sprite property; determines whether the contents of the sprite, the specified cast member, or a movie or cast file, or linked cast member is downloaded from the Internet and is available on the local disk (TRUE) or is not available (FALSE).

This property is useful only when streaming a movie or cast file. Movie streaming is activated by setting the Movie:Playback properties in the Modify menu to Play While Downloading Movie (default setting).

For a demonstration of the `mediaReady` member function, see the sample movie "Streaming Shockwave" in Director Help.

This property can be tested but not set.

### Example

This statement changes cast members when the desired cast member is downloaded and available locally:

```
if member("background").mediaReady = TRUE then
    sprite(2).memberNum = 10
    -- 10 is the number of cast member "background"
end if
```

### See also

`frameReady()`

## mediaStatus

### Syntax

```
sprite(whichSprite).mediaStatus
member(whichCastmember).mediaStatus
```

### Description

RealMedia sprite or cast member property; allows you to get a symbol representing the state of the RealMedia stream. This property can be tested but not set, and it is dynamic during playback.

This property can have the following values:

- `#closed` indicates that the RealMedia cast member is not active. The `mediaStatus` value remains `#closed` until playback is initiated.
- `#connecting` indicates that a connection to the RealMedia stream is being established.
- `#opened` indicates that a connection to the RealMedia stream has been established and is open. This is a transitory state that is very quickly followed by `#buffering`.
- `#buffering` indicates that the RealMedia stream is being downloaded into the playback buffer. When buffering is complete (`percentBuffered` equals 100), the RealMedia stream begins to play if the `pausedAtStart` property is `FALSE`. For more information, see `percentBuffered`.
- `#playing` indicates that the RealMedia stream is currently playing.
- `#seeking` indicates that play was interrupted by the `seek` command.
- `#paused` indicates that play has been interrupted, possibly by the user clicking the Stop button in the RealMedia viewer, or by a Lingo script invoking the `pause` method.
- `#error` indicates that the stream could not be connected, buffered, or played for some reason. The `lastError` property reports the actual error.

Depending on the cast member's state (`RealMedia`) value, a different `mediaStatus` property value is returned. Each `mediaStatus` value corresponds to only one state value. (For more information, see the description of the `state (RealMedia)` property.)

### Examples

The following examples show that the RealMedia element in sprite 2 and the cast member Real is playing.

```
put sprite(2).mediaStatus
-- #playing

put member("Real").mediaStatus
-- #playing
```

### See also

`state (RealMedia)`, `percentBuffered`, `lastError`

## member

### Syntax

```
member(whichCastmember).texture(whichTexture).member  
member(whichCastmember).model(whichModel).shader.texture.member  
member(whichCastmember).model(whichModel).shaderList\  
[shaderListIndex].textureList[textureListIndex].member
```

### Description

3D texture property; if the texture's type is `#fromCastMember`, this property indicates the cast member that is used as the source for a texture.

This property can be tested and set.

If the texture's type is `#importedFromFile`, this property value is void and cannot be set. If the texture's type is `#fromImageObject`, this property value is void, but it can be set.

### Example

This Lingo adds a new texture. The second statement shows that the cast member used to create the texture named `gbTexture` was member 16 of cast 1.

```
member("scene").newTexture("gbTexture", #fromCastmember, \  
    member(16, 1))  
put member("scene").texture("gbTexture").member  
-- (member 16 of castLib 1)
```

## member (keyword)

### Syntax

```
member whichCastMember  
member whichCastMember of castLib whichCast  
member(whichCastMember, whichCastLib)
```

### Description

Keyword; indicates that the object specified by *whichCastMember* is a cast member. If *whichCastMember* is a string, it is used as the cast member name. If *whichCastMember* is an integer, it is used as the cast member number.

When playing back a movie as an applet, refer to cast members by number rather than by name to improve the applet's performance.

The `member` keyword is a specific reference to both a `castLib` and a member within it if used alone:

```
put sprite(12).member  
-- (member 3 of castLib 2)
```

This property differs from the `memberNum` property of a `sprite`, which is always an integer designating position in a `castLib` but does not specify the `castLib`:

```
put sprite(12).memberNum  
-- 3
```

The number of a member is also an absolute reference to a particular member in a particular `castLib`:

```
put sprite(12).member.number  
-- 131075
```

### Examples

The following statement sets the `hilite` property of the button cast member named Enter Bid to `TRUE`:

```
member("Enter Bid").hilite = TRUE
```

This statement puts the name of sound cast member 132 into the variable `soundName`:

```
put member(132, "Viva Las Vegas").name
```

This statement checks the type of member Jefferson Portrait in the castLib Presidents:

```
memberType = member("Jefferson Portrait", "Presidents").type
```

This statement determines whether cast member 9 has a name assigned:

```
if member(9).name = EMPTY then exit
```

You can check for the existence of a member by testing for its number:

```
memberCheck = member("Epiphany").number  
if memberCheck = -1 then alert "Sorry, that member doesn't exist"
```

Alternatively, you can check for the existence of a member by testing for its type:

```
memberCheck = member("Epiphany").type  
if memberCheck = #empty then alert "Sorry, that member doesn't exist"
```

### See also

`memberNum`

## member (sound property)

### Syntax

```
sound(channelNum).member  
the member of sound(channelNum)
```

### Description

This read-only property is the sound cast member currently playing in sound channel *channelNum*. This property returns null if no sound is being played.

### Example

This statement displays the name of the member of the sound playing in sound channel 2 in the message window:

```
put sound(2).member  
-- (member 4 of castLib 1)
```

### See also

`getPlaylist()`, `queue()`

## member (sprite property)

### Syntax

```
sprite(whichSprite).member  
the member of sprite whichSprite
```

### Description

Sprite property; specifies a sprite's cast member and cast.

The `member sprite` property differs from the `memberNum sprite` property, which specifies only the sprite's number to identify its location in the cast but doesn't specify the cast itself. The `member sprite` property also differs from `mouseMember` and the obsolete `castNum sprite` properties, neither of which specifies the sprite's cast.

When assigning a sprite's `member` property, use one of the following formats:

- Specify the full member and cast description (`sprite(x).member = member(A, B)`).
- Specify the cast member name (`sprite(x).member = member ("MELODY ")`).
- Specify the unique integer that includes all cast libraries and corresponds to the `mouseMember` function (`sprite(x).member = 132`).

If you use only the cast member name, Director finds the first cast member that has that name in all current casts. If the name is duplicated in two casts, only the first name is used.

To specify a cast member by number when there are multiple casts, use the `memberNum sprite` property, which changes the member's position in its cast without affecting the sprite's cast (set the `memberNum` of sprite `x` to 132).

You can determine the `memberNum sprite` property from the `member sprite` property by using the phrase the number of the member of sprite `x`. You can also retrieve other cast member properties by using phrases such as the name of the member of sprite `x` or the rect of the member of sprite `x`.

The cast member assigned to a sprite channel is only one of that sprite's properties; other properties vary by the type of media element in that channel in the Score. For example, if you replace a bitmap with an unfilled shape by setting the `member sprite` property, the shape sprite's `lineSize` sprite property doesn't automatically change, and you probably won't see the shape.

Similar sprite property mismatches can occur if you change the member of a field sprite to a video. Although you can change all sprite properties through the `type sprite` property, it's generally more useful and predictable to replace cast members with similar cast members. For example, replace bitmap sprites with bitmap cast members.

This property can be tested and set.

### Examples

This statement assigns cast member 3 of cast number 4 to sprite 15:

```
sprite(15).member = member(3, 4)
```

The following handler uses the `mouseMember` function with the `sprite.member` property to find if the mouse is over a particular sprite:

```
on exitFrame
  MM = the mouseMember
  target = sprite(1).member
  if target = MM then put "above the hotspot"
  go the frame
end
```

### See also

`castLibNum`, `memberNum`

# memberNum

## Syntax

`sprite(whichSprite).memberNum`  
the memberNum of sprite *whichSprite*

## Description

Sprite property; identifies the position of the cast member (but doesn't identify the castLib) associated with the specified sprite *whichSprite*. Its value is the cast member number only; the cast member's cast is not specified.

The `memberNum` property is useful for switching cast members assigned to a sprite so long as the cast members are within the same cast. To switch among cast members in different casts, use the `member sprite` property. For the value set by Lingo to last beyond the current sprite, the sprite must be a puppet.

This property also is useful for exchanging cast members when a sprite is clicked to simulate the reversed image that appears when a standard button is clicked. You can also make some action in the movie depend on which cast member is assigned to a sprite.

When you set this property within a script while the playhead is not moving, be sure to use the `updateStage` command to redraw the Stage.

This property can be tested and set.

## Examples

The following statement switches the cast member assigned to sprite 3 to cast member number 35 in the same cast:

```
sprite(3).memberNum = 35
```

The following statement assigns the cast member Narrator to sprite 10 by setting `memberNum` sprite property to Narrator's cast number. Narrator is in the same cast as the sprite's current cast member.

```
sprite(10).memberNum = member("Narrator").number
```

The following handler swaps bitmaps when a button is clicked or rolled off. It assumes that the artwork for the Down button immediately follows the artwork for the Up button in the same cast.

```
on mouseDown
  upButton = sprite(the clickOn).memberNum
  downButton = upButton + 1
  repeat while the stillDown
    if rollover(the clickOn) then
      sprite(the clickOn).memberNum = downButton
    else
      sprite(the clickOn).memberNum = upButton
    end if
    updateStage
  end repeat
  if rollover (the clickOn) then put "The button was activated"
end
```

## See also

`castLib`, `member` (sprite property), `number` (cast member property), `member` (keyword)

## members

### See

number of members

## memorySize

### Syntax

the memorySize

### Description

System property; returns the total amount of memory allocated to the program, whether in use or free memory. This property is useful for checking minimum memory requirements. The value is given in bytes.

In Windows, the value is the total physical memory available; on the Macintosh, the value is the entire partition assigned to the application.

### Example

This statement checks whether the computer allocates less than 500K of memory and, if it does, displays an alert:

```
if the memorySize < 500 * 1024 then alert "There is not enough memory to run  
this movie."
```

### See also

freeBlock(), freeBytes(), ramNeeded(), size

## menu

### Syntax

```
menu: menuName  
itemName | script  
itemName | script  
...
```

or

```
menu: menuName  
itemName | script  
itemName | script  
...  
[more menus]
```

### Description

Keyword; in conjunction with the `installMenu` command, specifies the actual content of custom menus. Field cast members contain menu definitions; refer to them by the cast member name or number.

The `menu` keyword is followed immediately by a colon, a space, and the name of the menu. In subsequent lines, specify the menu items for that menu. You can set a script to execute when the user chooses an item by placing the script after the vertical bar symbol (`|`). A new menu is defined by the subsequent occurrence of the `menu` keyword.

**Note:** Menus are not available in Shockwave.

On the Macintosh, you can use special characters to define custom menus. These special characters are case sensitive. For example, to make a menu item bold, the letter *B* must be uppercase.

Special symbols should follow the item name and precede the vertical bar symbol (|). You can also use more than one special character to define a menu item. Using <B<U, for example, sets the style to Bold and Underline.

Avoid special character formatting for cross-platform movies because not all Windows computers support it.

Symbol	Example	Description
@	menu: @	*On the Macintosh, creates the Apple symbol and enables Macintosh menu bar items when you define an Apple menu.
!Å	!ÅEasy Select	*On the Macintosh, checks the menu with a check mark (Option+v).
<B	Bold<B	*On the Macintosh, sets the menu item's style to Bold.
<I	Italic<I	*On the Macintosh, sets the style to Italic.
<U	Underline<U	*On the Macintosh, sets the style to Underline.
<O	Outline<O	*On the Macintosh, sets the style to Outline.
<S	Shadow<S	*On the Macintosh, sets the style to Shadow.
	Open/O   go to frame "Open"	Associates a script with the menu item.
/	Quit/Q	Defines a command-key equivalent.
(	Save(	Disables the menu item.
( -	( -	Creates a disabled line in the menu.

\* identifies formatting tags that work only on the Macintosh.

### Example

This example is the text of a field cast member named CustomMenu2 which can be used to specify the content of a custom File menu. To install this menu, use “installMenu member(“CustomMenu2”)” while the movie is running. The Convert menu item runs the custom handler convertThis.

```
menu: File
Open/O | go to frame "Open"
Close/W | go to frame "Close"
Convert/C | convertThis
(-
Quit/Q | go to frame "Quit"
```

### See also

installMenu, name (menu property), name (menu item property), number (menu items), checkMark, enabled, script



# mesh (property)

## Syntax

```
member(whichCastmember).model(whichModel).\
meshdeform.mesh[index].meshProperty
```

## Description

3D command; allows access to the mesh properties of models that have the meshDeform modifier attached. When used as mesh.count this command returns the total number of meshes within the referenced model.

The properties of each mesh that are accessible are as follows:

- colorList allows you to get or set the list of colors used by the specified mesh.
- vertexList allows you to get or set the list of vertices used by the specified mesh.
- normalList allows you to get or set the list of normal vectors used by the specified mesh.
- textureCoordinateList allows you to get or set the texture coordinates used by the first texture layer of the specified mesh. To get or set the texture coordinates for any other texture layers in the specified mesh, use meshdeform.mesh[*index*].\texturelayer[*index*].textureCoordinateList.
- textureLayer[*index*] allows you get and set access to the properties of the specified texture layer.
- face[*index*] allows you to get or set the vertices, normals, texture coordinates, colors, and shaders used by the faces of the specified mesh.
- face.count allows you to obtain the total number of faces found within the specified mesh.

**Note:** For complete information about these properties, see the individual entries (listed in the “See also” section of this entry).

## Examples

The following Lingo adds the #meshDeform modifier to the model named thing1 and then displays the vertexList for the first mesh in the model named thing1.

```
member("newAlien").model("thing1").addModifier(#meshDeform)
put member("newAlien").model("thing1").meshDeform.mesh[1].vertexList
-- [vector(239.0, -1000.5, 27.4), vector\
   (162.5, -1064.7, 29.3), vector(115.3, -1010.8, -40.6),
   vector(239.0, -1000.5, 27.4), vector(115.3, -1010.8, -40.6),
   vector(162.5, -1064.7, 29.3), vector(359.0, -828.5, -46.3),
   vector(309.9, -914.5, -45.3)]
```

The following statement displays the number of meshes found within the model named “Aircraft”.

```
put member("world").model("Aircraft").meshDeform.mesh.count
-- 4
```

## See also

meshDeform (modifier), colorList, textureCoordinateList, textureLayer, normalList, vertexList (mesh deform), face

## meshDeform (modifier)

### Syntax

`member(whichCastmember).model(whichModel).meshDeform.propertyName`

### Description

3D modifier; allows control over the various aspects of the referenced model's mesh structure. Once you have added the `#meshDeform` modifier (using the `addModifier` command) to a model you have access to the following properties of the `#meshDeform` modifier:

**Note:** For more detailed information about the following properties see the individual property entries referenced in the see also section of this entry.

- `face.count` returns the total number of faces in the referenced model.
- `mesh.count` returns the number of meshes in the referenced model.
- `mesh[index]` allows access to the properties of the specified mesh.

### Examples

The following statement displays the number of faces in the model named `gbFace`:

```
put member("3D World").model("gbFace").meshDeform.face.count
-- 432
```

The following statement displays the number of meshes in the model named `gbFace`:

```
put member("3D World").model("gbFace").meshDeform.mesh.count
-- 2
```

The following statement displays the number of faces in the second mesh of the model named `gbFace`:

```
put member("3D World").model("gbFace").meshDeform.mesh[2].face.count
-- 204
```

### See also

`mesh (property), addModifier`

## milliseconds

### Syntax

`the milliseconds`

### Description

System property; returns the current time in milliseconds (1/1000 of a second). Counting begins from the time the computer is started.

### Examples

This statement converts milliseconds to seconds and minutes by dividing the number of milliseconds by 1000 and dividing that result by 60, and then sets the variable `currentMinutes` to the result:

```
currentSeconds = milliseconds/1000
currentMinutes = currentseconds/60
```

The resolution accuracy of the count is machine and operating system dependent.

This handler counts the milliseconds and posts an alert if you've been working too long:

```
on idle
  if the milliseconds > 1000 * 60 * 60 * 4 then
    alert "Take a break"
  end if
end
```

**See also**

ticks, time(), timer

## min

**Syntax**

```
list.min
min(list)
min(a1, a2, a3...)
```

**Description**

Function; specifies the minimum value in the list specified by *list*.

**Example**

The following handler assigns the variable `vLowest` the minimum value in the list `bids`, which consists of [#Castle:600, #Shields:750, #Wang:230]. The result is then inserted in the content of the field `cast` member `Sorry`:

```
on findLowest bids
  vLowest = bids.min()
  member("Sorry").text = \
    "We're sorry, your bid of $" & vLowest && "is not a winner!"
end
```

**See also**

max()

## minSpeed

**Syntax**

```
member(whichCastmember).modelResource(whichModelResource).
  emitter.minSpeed
```

**Description**

3D property; when used with a model resource whose type is `#particle`, allows you to get and set the minimum speed at which particles are emitted. Each particle's initial velocity is randomly selected between the emitter's `minSpeed` and `maxSpeed` properties.

The value is a floating-point number and must be greater than 0.0.

**Example**

In this example, `ThermoSystem` is a model resource of the type `#particle`. This statement sets the `minSpeed` property of `ThermoSystem` to 5, which causes the slowest particles of `ThermoSystem` to move somewhat slowly. Within a given particle system, the slower a particle moves, the shorter the distance it will travel.

```
member("Fires").modelResource("ThermoSystem").emitter.\
  minSpeed = 5
```

**See also**

maxSpeed, emitter

## missingFonts

### Syntax

`member(textCastMember).missingFonts`

### Description

Text cast member property; this property contains a list of the names of the fonts that are referenced in the text, but not currently available on the system.

This allows the developer to determine during run time if a particular font is available or not.

This property can be tested but not set.

### See also

`substituteFont`

## mod

### Syntax

`integerExpression1 mod integerExpression2`

### Description

Math operator; performs the arithmetic modulus operation on two integer expressions. In this operation, *integerExpression1* is divided by *integerExpression2*.

The resulting value of the entire expression is the integer remainder of the division. It always has the sign of *integerExpression1*.

This is an arithmetic operator with a precedence level of 4.

### Examples

This statement divides 7 by 4 and then displays the remainder in the Message window:

```
put 7 mod 4
```

The result is 3.

The following handler sets the ink effect of all odd-numbered sprites to `copy`, which is the ink effect specified by the number 0. First the handler checks whether the sprite in the variable `mySprite` is an odd-numbered sprite by dividing the sprite number by 2 and then checking whether the remainder is 1. If the remainder is 1, the result for an odd-numbered number, the handler sets the ink effect to `copy`.

```
on setInk
  repeat with mySprite = 1 to the lastChannel
    if (mySprite mod 2) = 1 then
      sprite(mySprite).ink = 0
    else
      sprite(mySprite).ink = 8
    end if
  end repeat
end setInk
```

This handler regularly cycles a sprite's cast member among a number of bitmaps:

```
on exitFrame
  global gCounter
  -- These are sample values for bitmap cast member numbers
  theBitmaps = [2,3,4,5,6,7]
  -- Specify which sprite channel is affected
  theChannel = 1
  -- This cycles through the list
  gCounter = 1 + (gCounter mod theBitmaps.count)
  sprite(theChannel).memberNum = theBitmaps[gCounter]
  go the frame
end
```

## modal

### Syntax

```
window "window".modal
the modal of window "window"
```

### Description

Window property; specifies whether movies can respond to events that occur outside the window specified by *window*.

- When the `modal` window property is `TRUE`, movies cannot respond to events outside the window.
- When the `modal` window property is `FALSE`, movies can respond to events outside the window.

Setting the `modal` window property to `TRUE` lets you make a specific movie in a window the only movie that the user can interact with.

Be aware that this property works even in the authoring environment. If you set the `modal` window property to `TRUE`, you will not be able to interact with the Director windows either.

You can always close a window that is modal by using Control+Alt+period (Windows) or Command+period (Macintosh).

### Example

This statement lets movies respond to events outside of the Tool Panel window:

```
window("Tool Panel").modal = FALSE
```

## mode (emitter)

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
  emitter.mode
```

### Description

3D property; when used with a model resource whose type is `#particle`, allows you to both get and set the `mode` property of the resource's particle emitter.

This property can have the value `#burst` or `#stream` (default). A mode value of `#burst` causes all particles to be emitted at the same time, while a value of `#stream` causes a group of particles to be emitted at each frame. The number of particles emitted in each frame is determined using the following equation:

```
particlesPerFrame = resourceObject.emitter.numParticles \
    (resourceObject.lifetime x millisecondsPerRenderedFrame)
```

#### Example

In this example, `ThermoSystem` is a model resource of the type `#particle`. This statement sets the `emitter.mode` property of `ThermoSystem` to `#burst`, which causes the particles of `ThermoSystem` to appear in bursts. To create a single burst of particles, set `emitter.mode = #burst` and `emitter.loop = 0`.

```
member("Fires").modelResource("ThermoSystem").emitter.mode = #burst
```

#### See also

`emitter`

## mode (collision)

#### Syntax

```
member(whichCastmember).model(whichModel).collision.mode
```

#### Description

3D collision modifier property; indicates the geometry to be used in the collision detection algorithm. Using simpler geometry such as the bounding sphere leads to better performance. The possible values for this property are:

- `#mesh` uses the actual mesh geometry of the model's resource. This gives one-triangle precision and is usually slower than `#box` or `#sphere`.
- `#box` uses the bounding box of the model. This is useful for objects that can fit more tightly in a box than in a sphere, such as a wall.
- `#sphere` is the fastest mode, because it uses the bounding sphere of the model. This is the default value for this property.

#### Example

These statements add the collision modifier to the model named `your Model` and set the mode property to `#mesh`:

```
member("3d").model("yourModel").addModifier(#collision)
member("3d").model("yourModel").collision.mode = #mesh
```

## model

#### Syntax

```
member(whichCastmember).model(whichModel)
member(whichCastmember).model[index]
member(whichCastmember).model.count
member(whichCastmember).model(whichModel).propertyName
member(whichCastmember).model[index].propertyName
```

### Description

3D command; returns the model found within the referenced cast member that has the name specified by *whichModel*, or is found at the index position specified by *index*. If no model exists for the specified parameter, the command returns `void`. As `model.count`, the command returns the number of models found within the referenced cast member. This command also allows access to the specified model's properties.

Model name comparisons are not case-sensitive. The index position of a particular model may change when objects at lower index positions are deleted.

If no model is found that uses the specified name or no model is found at the specified index position then this command returns `void`.

### Examples

This statement stores a reference to the model named Player Avatar in the variable `thismodel`:

```
thismodel = member("3DWorld").model("Player Avatar")
```

This statement stores a reference to the eighth model of the cast member named 3DWorld in the variable `thismodel`.

```
thismodel = member("3DWorld").model[8]
```

This statement shows that there are four models in the member of sprite 1.

```
put sprite(1).member.model.count  
-- 4
```

## modelA

### Syntax

```
collisionData.modelA
```

### Description

3D `collisionData` property; indicates one of the models involved in a collision, the other model being `modelB`.

The `collisionData` object is sent as an argument with the `#collideWith` and `#collideAny` events to the handler specified in the `registerForEvent`, `registerScript`, and `setCollisionCallback` commands.

The `#collideWith` and `#collideAny` events are sent when a collision occurs between models to which collision modifiers have been added. The `resolve` property of the models' modifiers must be set to `TRUE`.

This property can be tested but not set.

### Example

This example has three parts. The first part is the first line of code, which registers the `#putDetails` handler for the `#collideAny` event. The second part is the `#putDetails` handler. When two models in the cast member named `MyScene` collide, the `#putDetails` handler is called and the `collisionData` argument is sent to it. This handler displays the `modelA` and `modelB` properties of the `collisionData` object in the message window. The third part of the example shows the results from the message window. These show that the model named `GreenBall` was `modelA` and the model named `YellowBall` was `modelB` in the collision.

```
member("MyScene").registerForEvent(#collideAny, #putDetails, 0)
on putDetails me, collisionData
    put collisionData.modelA
    put collisionData.modelB
end
-- model("GreenBall")
-- model("YellowBall")
```

### See also

`registerScript()`, `registerForEvent()`, `sendEvent`, `modelB`, `setCollisionCallback()`

## modelB

### Syntax

`collisionData.modelB`

### Description

3D `collisionData` property; indicates one of the models involved in a collision, the other model being `modelA`.

The `collisionData` object is sent as an argument with the `#collideWith` and `#collideAny` events to the handler specified in the `registerForEvent`, `registerScript`, and `setCollisionCallback` commands.

The `#collideWith` and `#collideAny` events are sent when a collision occurs between models to which collision modifiers have been added. The `resolve` property of the models' modifiers must be set to `TRUE`.

This property can be tested but not set.

### Example

This example has three parts. The first part is the first line of code, which registers the `#putDetails` handler for the `#collideAny` event. The second part is the `#putDetails` handler. When two models in the cast member named `MyScene` collide, the `#putDetails` handler is called and the `collisionData` argument is sent to it. This handler displays the `modelA` and `modelB` properties of the `collisionData` object in the message window. The third part of the example shows the results from the message window. These show that the model named `GreenBall` was `modelA` and the model named `YellowBall` was `modelB` in the collision.

```
member("MyScene").registerForEvent(#collideAny, #putDetails, 0)
on putDetails me, collisionData
    put collisionData.modelA
    put collisionData.modelB
end
-- model("GreenBall")
-- model("YellowBall")
```

### See also

`registerScript()`, `registerForEvent()`, `sendEvent`, `modelA`, `collisionNormal`, `setCollisionCallback()`



## modelResource

### Syntax

```
member(whichCastmember).modelResource(whichModelResource)
member(whichCastmember).modelResource[index]
member(whichCastmember).modelResource.count
member(whichCastmember).modelResource(whichModelResource).\
    propertyName
member(whichCastmember).modelResource[index].propertyName
```

### Description

3D command; returns the model resource found within the referenced cast member that has the name specified by *whichModelResource*, or is found at the index position specified by the *index* parameter. If no model resource exists for the specified parameter, the command returns void. As `modelResource.count`, the command returns the number of model resources found within the referenced cast member. This command also allows access to the specified model resource's properties.

Model resource name string comparisons are not case-sensitive. The index position of a particular model resource may change when objects at lower index positions are deleted.

### Examples

This statement stores a reference to the model resource named HouseA in the variable `thismodelResource`.

```
thismodelResource = member("3DWorld").modelResource("HouseA")
```

This statement stores a reference to the fourteenth model resource of the cast member named 3DWorld in the variable `thismodelResource`.

```
thismodelResource = member("3DWorld").modelResource[14]
```

This statement shows that there are ten model resources in the member of sprite 1.

```
put sprite(1).member.modelResource.count
--10
```

## modelsUnderLoc

### Syntax

```
member(whichCastmember).camera(whichCamera).modelsUnderLoc\
    (pointWithinSprite {, maxNumberOfModels, levelOfDetail})
```

### Description

3D command; returns a list of models found under the point specified by *pointWithinSprite* within the rect of a sprite using the referenced camera. The location *pointWithinSprite* is relative to the upper left corner of the sprite, in pixels.

The optional *maxNumberOfModels* parameter allows you to limit the length of the returned list. If this parameter isn't specified, the command returns a list containing references for all of the models found under the specified point.

The optional *levelOfDetail* parameter allows you to specify the level of detail of the information returned. The *levelOfDetail* parameter can have the following values:

`#simple` returns a list containing references to the models found under the point. This is the default setting.

`#detailed` returns a list of property lists, each representing an intersected model. Each property list will have the following properties:

- `#model` is a reference to the intersected model object.
- `#distance` is the distance from the camera to the point of intersection with the model.
- `#isectPosition` is a vector representing the world space position of the point of intersection.
- `#isectNormal` is the world space normal vector to the mesh at the point of intersection.
- `#meshID` is the meshID of the intersected mesh, which can be used as an index into the mesh list of the `meshDeform` modifier.
- `#faceID` is the face ID of the intersected face, which can be used as an index into the face list of the `meshDeform` modifier.
- `#vertices` is a three-element list of vectors that represent the world space positions of the vertices of the intersected face.
- `#uvCoord` is a property list with properties `#u` and `#v` that represent the u and v barycentric coordinates of the face.

Within the returned list, the first model listed is the one closest to the viewer and the last model listed is the furthest from the viewer.

Only one intersection (the closest intersection) is returned per model.

The command returns an empty list if there are no models found under the specified point.

#### Example

The first line in this handler translates the location of the cursor from a point on the Stage to a point within sprite 5. The second line uses the `modelsUnderLoc` command to obtain the first three models found under that point. The third line displays the returned detailed information about the models in the message window.

```
on mouseUp
    pt = the mouseLoc - point(sprite(5).left, sprite(5).top)
    m = sprite(5).camera.modelsUnderLoc(pt, 3, #detailed)
    put m
end
```

#### See also

`modelsUnderRay`, `modelUnderLoc`

## modelsUnderRay

#### Syntax

```
member(whichCastmember).modelsUnderRay(locationVector, directionVector {, maxNumberOfModels, levelOfDetail})
```

#### Description

3D command; returns a list of models found under a ray drawn from the position specified by *locationVector* and pointing in the direction of *directionVector*, with both vectors being specified in world-relative coordinates.

The optional *maxNumberOfModels* parameter allows you to limit the length of the returned list. If this parameter isn't specified, the command returns a list containing references for all of the models found under the specified ray.

The optional *levelOfDetail* parameter allows you to specify the level of detail of the information returned. The *levelOfDetail* parameter can have the following values:

*#simple* returns a list containing references to the models found under the point. This is the default setting.

*#detailed* returns a list of property lists, each representing an intersected model. Each property list will have the following properties:

- *#model* is a reference to the intersected model object.
- *#distance* is the distance from the world position specified by *locationVector* to the point of intersection with the model.
- *#isectPosition* is a vector representing the world space position of the point of intersection.
- *#isectNormal* is the world space normal vector to the mesh at the point of intersection.
- *#meshID* is the meshID of the intersected mesh which can be used to index into the mesh list of the *meshDeform* modifier.
- *#faceID* is the face ID of the intersected face which can be used to index into the face list of the *meshDeform* modifier.
- *#vertices* is a 3-element list of vectors that represent the world space positions of the vertices of the intersected face.
- *#uvCoord* is a property list with properties *#u* and *#v* that represent the u and v barycentric coordinates of the face.

Within the returned list, the first model listed is the one closest to the position specified by *locationVector* and the last model listed is the furthest from that position.

Only one intersection (the closest intersection) is returned per model.

The command returns an empty list if there are no models found under the specified ray.

### Example

This statement displays the detailed information for a model intersected by a ray drawn from the position vector (0, 0, 300) and pointing down the -z axis:

```
put member("3d").modelsUnderRay(vector(0, 0, 300), vector(0, 0, -\
1), 3, #detailed)
-- [[#model: model("mSphere"), #distance: 275.0000, \
#isectPosition: vector( 0.0000, 0.0000, 25.0000 ), #isectNormal: \
vector( -0.0775, 0.0161, 0.9969 ), #meshID: 1, #faceID: 229, \
#vertices: [vector( 0.0000, 0.0000, 25.0000 ), vector( -3.6851, \
1.3097, 24.6922 ), vector( -3.9017, 0.2669, 24.6922 )], \
#uvCoord: [#u: 0.0000, #v: 0.0000]]]
```

### See also

*modelsUnderLoc*, *modelUnderLoc*

## modelUnderLoc

### Syntax

```
member(whichCastmember).camera(whichCamera).\  
modelUnderLoc(pointWithinSprite)
```

### Description

3D command; returns a reference to the first model found under the point specified by *pointWithinSprite* within the rect of a sprite using the referenced camera. The location *pointWithinSprite* is relative to the upper left corner of the sprite, in pixels.

This command returns `void` if there is no model found under the specified point.

For a list of all of the models found under a specified point, and detailed information about them, see `modelsUnderLoc`.

### Example

The first line in this handler translates the location of the cursor from a point on the Stage to a point within sprite 5. The second line determines the first model under that point. The third line displays the result in the message window.

```
on mouseUp  
    pt = the mouseLoc - point(sprite(5).left, sprite(5).top)  
    m = sprite(5).camera.modelUnderLoc(pt)  
    put m  
end
```

### See also

`modelsUnderLoc`, `modelsUnderRay`

## modified

### Syntax

```
member(whichCastMember).modified  
the modified of member whichCastMember
```

### Description

Cast member property; indicates whether the cast member specified by *whichCastMember* has been modified since it was read from the movie file.

- When the `modified` member property is `TRUE` (1), the cast member has been modified since it was read from the movie file.
- When the `modified` member property is `FALSE` (0), the cast member has not been modified since it was read from the movie file.

The `modified` member function always returns `FALSE` for a cast member in a movie that plays back as an applet.

### Example

This statement tests whether the cast member `Introduction` has been modified since it was read from the movie file:

```
return member("Introduction").modified
```

## modifiedBy

### Syntax

*member.modifiedBy*  
the *modifiedBy* of *member*

### Description

Cast member property; records the name of the user who last edited the cast member, taken from the user name information provided during Director installation. You can change this information in the Director General Preferences dialog box.

This property can be tested but not set. It is useful for tracking and coordinating Director projects with more than one author, and may also be viewed in the Property inspector's Member tab.

### Example

This statement displays the name of the person who last modified cast member 1:

```
put member(1).modifiedBy
-- "Sam Sein"
```

### See also

comments, creationDate, modifiedDate

## modifiedDate

### Syntax

*member.modifiedDate*  
the *modifiedDate* of *member*

### Description

Cast member property; indicates the date and time that the cast member was last changed, using the system time on the authoring computer. This property is useful for tracking and coordinating Director projects.

This property can be tested but not set. It can also be viewed in the Property inspector's Member tab and the Cast window list view.

### Example

This statement displays the date of the last change to cast member 1:

```
put member(1).modifiedDate
-- date( 1999, 12, 8 )
```

### See also

comments, creationDate, modifiedBy

## modifier

### Syntax

*member(whichCastmember).model(whichModel).modifier*  
*member(whichCastmember).model(whichModel).modifier.count*

### Description

3D model property; returns a list of modifiers that are attached to the specified model. As *modifier.count*, the command returns the number of modifiers attached to the model.

Note that if both the `toon` and `inker` modifiers are applied to a model, only the first one that was added to the model is returned.

This property can be tested but not set. Use the `addModifier` and `removeModifier` commands to add and remove modifiers from models.

#### Example

This statement shows which modifiers are attached to the model named `Juggler`:

```
put member("ParkScene").model("Juggler").modifier
-- [#bonesPlayer, #lod]
```

#### See also

`modifier[]`, `modifiers`, `addModifier`, `removeModifier`

## modifier[]

#### Syntax

`member(whichCastmember).model(whichModel).modifier[index]`

#### Description

3D model property; returns the type of the modifier found at the position specified by *index* within the model's attached modifier list. The value returned is a symbol.

If no modifier is found at the specified position then this property's value is void.

To obtain information about a model's attached modifier list use the `modifier` property.

Direct access into an attached modifier's properties is not supported through the use of this command.

#### Example

```
put member("3d world").model("box").modifier[1]
-- #lod
```

#### See also

`modifier`, `modifiers`, `addModifier`, `removeModifier`

## modifiers

#### Syntax

`getRendererServices().modifiers`

#### Description

Global 3D property; returns a list of modifiers available to models within 3D cast members.

#### Example

This statement returns the list of all currently available modifiers:

```
put getRendererServices().modifiers
-- [#collision, #bonesPlayer, #keyFramePlayer, #toon, #lod, \
    #meshDeform, #sds, #inker]
```

#### See also

`getRendererServices()`, `addModifier`

## mostRecentCuePoint

### Syntax

```
sprite(whichSprite).mostRecentCuePoint  
the mostRecentCuePoint of sprite whichSprite  
sound(channelNum).mostRecentCuePoint  
the mostRecentCuePoint of sound channelNum
```

### Description

Cast member, sound channel, and sprite property; for sound cast members, QuickTime digital video, and Xtra extensions that support cue points, indicates the number that identifies the most recent cue point passed in the sprite or sound. The value is the cue point's ordinal number. If no cue points have been passed, the value is 0.

For sound channels, the return value is for the sound cast member currently playing in the sound channel.

Shockwave Audio (SWA) sounds can appear as sprites in sprite channels, but they play sound in a sound channel. It is recommended that you refer to SWA sound sprites by their sprite channel number rather than their sound channel number.

### Examples

This statement tells the Message window to display the number for the most recent cue point passed in the sprite in sprite channel 1:

```
put sprite(1).mostRecentCuePoint
```

This statement returns the ordinal number of the most recently passed cue point in the currently playing sound in sound channel 2:

```
put sound(2).mostRecentCuePoint
```

### See also

`cuePointNames`, `isPastCuePoint()`, `cuePointTimes`, `on cuePassed`

## motion

### Syntax

```
member(whichCastmember).motion(whichMotion)  
member(whichCastmember).motion[index]  
member(whichCastmember).motion.count
```

### Description

3D command; returns the motion found within the referenced cast member that has the name specified by *whichMotion*, or is found at the index position specified by the *index*. As `motion.count`, this property returns the total number of motions found within the cast member.

Object name string comparisons are not case-sensitive. The index position of a particular motion may change when objects at lower index positions are deleted.

If no motion is found that uses the specified name or no motion is found at the specified index position then this command returns void.

### Examples

```
thisMotion = member("3D World").motion("Wing Flap")
thisMotion = member("3D World").motion[7]
put member("scene").motion.count
-- 2
```

### See also

duration (3D), map (3D)

## motionQuality

### Syntax

sprite(*whichQTVRSprite*).motionQuality  
motionQuality of sprite *whichQTVRSprite*

### Description

QuickTime VR sprite property; the codec quality used when the user clicks and drags the QuickTime VR sprite. The property's value can be #minQuality, #maxQuality, or #normalQuality.

This property can be tested and set.

### Example

This statement sets the motionQuality of sprite 1 to #minQuality.

```
sprite(1).motionQuality = #minQuality
```

## mouseChar

### Syntax

the mouseChar

### Description

System property; used for field sprites, contains the number of the character that is under the pointer when the property is called. The count is from the beginning of the field. If the mouse pointer is not over a field or is in the gutter of a field, the result is -1.

The value of the mouseChar property can change in a handler or repeat loop. If a handler or repeat loop uses this property multiple times, it's usually a good idea to call the property once and assign its value to a local variable.

### Examples

This statement determines whether the pointer is over a field sprite and changes the content of the field cast member Instructions to "Please point to a character." when it is:

```
if the mouseChar = -1 then
    member("Instructions").text = "Please point to a character."
```

This statement assigns the character under the pointer in the specified field to the variable currentChar:

```
currentChar = member(the mouseMember).char[the mouseChar]
```



This handler highlights the character under the pointer when the sprite contains a text cast member:

```
property spriteNum

on mouseWithin me
  if sprite(spriteNum).member.type = #field then
    MC = the mousechar
    if MC < 1 then exit    -- if over a border, final line, etc.
    hilite char MC of field sprite(spriteNum).member
    else alert "Sorry, 'hilite' and 'mouseChar' are for fields."
  end
end
```

**See also**

mouseItem, mouseLine, mouseWord, char...of, number (characters)

## on mouseDown (event handler)

**Syntax**

```
on mouseDown
  statement(s)
end
```

**Description**

System message and event handler; contains statements that run when the mouse button is pressed.

When the mouse button is pressed, Lingo searches the following locations, in order, for an `on mouseDown` handler: primary event handler, sprite script, cast member script, frame script, and movie script. Lingo stops searching when it reaches the first location that has an `on mouseDown` handler, unless the handler includes the `pass` command to explicitly pass the `mouseDown` message on to the next location.

To have the same response throughout the movie when pressing the mouse button, set `mouseDownScript` or put a `mouseDown` handler in a Movie script.

The `on mouseDown` event handler is a good place to put Lingo that flashes images, triggers sound effects, or makes sprites move when the user presses the mouse button.

Where you place an `on mouseDown` handler can affect when it runs.

- To apply the handler to a specific sprite, put it in a sprite script.
- To apply the handler to a cast member in general, put it in a cast member script.
- To apply the handler to an entire frame, put it in a frame script.
- To apply the handler throughout the entire movie, put it in a movie script.

You can override an `on mouseDown` handler by placing an alternative `on mouseDown` handler in a location that Lingo checks before it gets to the handler you want to override. For example, you can override an `on mouseDown` handler assigned to a cast member by placing an `on mouseDown` handler in a sprite script.

If used in a behavior, this event is passed the sprite script or frame script reference `me`.

### Examples

This handler checks whether the user clicks anywhere on the Stage and sends the playhead to another frame if a click occurs:

```
on mouseDown
  if the clickOn = 0 then go to frame "AddSum"
end
```

This handler, assigned to a sprite script, plays a sound when the sprite is clicked:

```
on mouseDown
  puppetSound "Crickets"
end
```

### See also

`clickOn`, `mouseDownScript`, `mouseUpScript`

## the `mouseDown` (system property)

### Syntax

the `mouseDown`

### Description

System property; indicates whether the mouse button is currently being pressed (TRUE) or not (FALSE).

The Director player for Java doesn't update the `mouseDown` property when Lingo is in a repeat loop. If you try to test `mouseDown` inside a repeat loop in an applet, the applet hangs.

### Example

This handler causes the movie to beep until the user clicks the mouse:

```
on enterFrame
  repeat while the mouseDown = FALSE
    beep
  end repeat
end
```

This statement instructs Lingo to exit the repeat loop or handler it is in when the user clicks the mouse:

```
if the mouseDown then exit
```

### See also

`mouseH`, the `mouseUp` (system property), `mouseV`, `on mouseDown` (event handler), `on mouseUp` (event handler)

## `mouseDownScript`

### Syntax

the `mouseDownScript`

### Description

System property; specifies the Lingo that is executed when the mouse button is pressed. The Lingo is written as a string, surrounded by quotation marks and can be a simple statement or a calling script for a handler. The default value is `EMPTY`, which means that the `mouseDownScript` property has no Lingo assigned to it.

When the mouse button is pressed and the `mouseDownScript` property is defined, Lingo executes the instructions specified for the `mouseDownScript` property first. No other `on mouseDown` handlers are executed, unless the instructions include the `pass` command so that the `mouseDown` message can be passed to other objects in the movie.

Setting the `mouseDownScript` property performs the same function as the `when keyDown then` command in earlier versions of Director.

To turn off the instructions you've specified for the `mouseDownScript` property, use the statement `set the mouseDownScript to empty`.

This property can be tested and set.

### Example

In this statement, when the user clicks the mouse button, the playhead always branches to the next marker in the movie:

```
the mouseDownScript = "go next"
```

In this statement, when the user clicks anywhere on the Stage, the computer beeps:

```
the mouseDownScript = "if the clickOn = 0 then beep"
```

The following statement sets `mouseDownScript` to the custom handler *myCustomHandler*. A Lingo custom handler must be enclosed in quotation marks when used with the `mouseDownScript` property.

```
the mouseDownScript = "myCustomHandler"
```

### See also

`stopEvent`, `mouseUpScript`, `on mouseDown (event handler)`, `on mouseUp (event handler)`

## on mouseEnter

### Syntax

```
on mouseEnter  
    statement(s)  
end
```

### Description

System message and event handler; contains statements that run when the mouse pointer first contacts the active area of the sprite. The mouse button does not have to be pressed.

If the sprite is a bitmap cast member with matte ink applied, the active area is the portion of the image that is displayed; otherwise, the active area is the sprite's bounding rectangle.

If used in a behavior, this event is passed the sprite script or frame script reference `me`.

### Example

This example is a simple button behavior that switches the bitmap of the button when the mouse rolls over and then off the button:

```
property spriteNum
on mouseEnter me
    -- Determine current cast member and switch to next in cast
    currentMember = sprite(spriteNum).member.number
    sprite(spriteNum).member = currentMember + 1
end
on mouseLeave me
    -- Determine current cast member and switch to previous in cast
    currentMember = sprite(spriteNum).member.number
    sprite(spriteNum).member = currentMember - 1
end
```

### See also

on mouseLeave, on mouseWithin

## mouseH

### Syntax

```
the mouseH
mouseH()
```

### Description

System property and function; indicates the horizontal position of the mouse pointer. The value of `mouseH` is the number of pixels the cursor is positioned from the left edge of the Stage.

The `mouseH` function is useful for moving sprites to the horizontal position of the mouse pointer and checking whether the pointer is within a region of the Stage. Using the `mouseH` and `mouseV` functions together, you can determine the cursor's exact location.

The Director player for Java doesn't update the `mouseH` function when Lingo is in a repeat loop.

This function can be tested but not set.

### Examples

This handler moves sprite 10 to the mouse pointer location and updates the Stage when the user clicks the mouse button:

```
on mouseDown
    sprite(10).locH = the mouseH
    sprite(10).locV = the mouseV
    updateStage
end
```

This statement tests whether the pointer is more than 10 pixels to the right or left of a starting point and, if it is, sets the variable `Far` to `TRUE`:

```
if abs(mouseH() - startH) > 10 then
    draggedEnough = TRUE
```

### See also

locH, locV, mouseV, mouseLoc

# mouseItem

## Syntax

the mouseItem

## Description

System property; contains the number of the item under the pointer when the function is called and the pointer is over a field sprite. (An item is any sequence of characters delimited by the current delimiter as set by the `itemDelimiter` property.) Counting starts at the beginning of the field. If the mouse pointer is not over a field, the result is -1.

The value of the `mouseItem` property can change in a handler or repeat loop. If a handler or repeat loop relies on the initial value of `mouseItem` when the handler or repeat loop begins, call the property once and assign its value to a local variable.

## Examples

This statement determines whether the pointer is over a field sprite and changes the content of the field cast member Instructions to "Please point to an item." when it is not:

```
if the mouseItem = -1 then
    member("Instructions") = "Please point to an item."
end if
```

This statement assigns the item under the pointer in the specified field to the variable `currentItem`:

```
currentItem = member(mouseMember).item(mouseItem)
```

This handler highlights the item under the pointer when the mouse button is pressed:

```
on mouseDown
    thisField = sprite(the clickOn).member
    if the mouseItem < 1 then exit
    lastItem = 0
    repeat while the stillDown
        MI = the mouseItem
        if MI < 1 then next repeat
        if MI <> lastItem then
            thisField.item[MI].hilite()
            lastItem = MI
        end if
    end repeat
end
```

## See also

`item...of`, `mouseChar`, `mouseLine`, `mouseWord`, `number (items)`, `itemDelimiter`

## on mouseLeave

### Syntax

```
on mouseLeave
    statement(s)
end
```

### Description

System message and event handler; contains statements that run when the mouse leaves the active area of the sprite. The mouse button does not have to be pressed.

If the sprite is a bitmap cast member with the matte ink applied, the active area is the portion of the image that is displayed; otherwise, the active area is the sprite's bounding rectangle.

If used in a behavior, this event is passed the sprite script or frame script reference `me`.

### Example

This statement shows a simple button behavior that switches the bitmap of the button when the mouse pointer rolls over and then back off the button:

```
property spriteNum
on mouseEnter me
    -- Determine current cast member and switch to next in cast
    currentMember = sprite(spriteNum).member.number
    sprite(spriteNum).member = currentMember + 1
end
on mouseLeave me
    -- Determine current cast member and switch to previous in cast
    currentMember = sprite(spriteNum).member.number
    sprite(spriteNum).member = currentMember - 1
end
```

### See also

`on mouseEnter`, `on mouseWithin`

## mouseLevel

### Syntax

```
sprite(whichQuickTimeSprite).mouseLevel
the mouseLevel of sprite whichQuickTimeSprite
```

### Description

QuickTime sprite property; controls how Director passes mouse clicks on a QuickTime sprite to QuickTime. The ability to pass mouse clicks within the sprite's bounding rectangle can be useful for interactive media such as QuickTime VR. The `mouseLevel` sprite property can have these values:

- `#controller`—Passes clicks only on the movie controller to QuickTime. Director responds only to mouse clicks that occur outside the controller. This is the standard behavior for QuickTime sprites other than QuickTime VR.
- `#all`—Passes all mouse clicks within the sprite's bounding rectangle to QuickTime. No clicks pass to other Lingo handlers.
- `#none`—Does not pass any mouse clicks to QuickTime. Director responds to all mouse clicks.
- `#shared`—Passes all mouse clicks within a QuickTime VR sprite's bounding rectangle to QuickTime and then passes these events to Lingo handlers. This is the default value for QuickTime VR.

This property can be tested and set.

### Example

This frame script checks to see if the name of the QuickTime sprite in channel 5 contains the string “QTVR.” If it does, this script sets `mouseLevel` to `#all`; otherwise, it sets `mouseLevel` to `#none`.

```
on prepareFrame
  if sprite(5).member.name contains "QTVR" then
    sprite(5).mouseLevel = #all
  else
    sprite(5).mouseLevel = #none
  end if
end
```

## mouseLine

### Syntax

the `mouseLine`

### Description

System property; contains the number of the line under the pointer when the property is called and the pointer is over a field sprite. Counting starts at the beginning of the field; a line is defined by Return delimiter, not by the wrapping at the edge of the field. When the mouse pointer is not over a field sprite, the result is -1.

The value of the `mouseLine` property can change in a handler or repeat loop. If a handler or repeat loop uses this property multiple times, it's usually a good idea to call the property once and assign its value to a local variable.

### Examples

This statement determines whether the pointer is over a field sprite and changes the content of the field cast member Instructions to “Please point to a line.” when it is not:

```
if the mouseLine = -1 then
  member("Instructions").text = "Please point to a line."
```

This statement assigns the contents of the line under the pointer in the specified field to the variable `currentLine`:

```
currentLine = member(the mouseMember).line[the mouseLine]
```

This handler highlights the line under the pointer when the mouse button is pressed:

```
on mouseDown
  thisField = sprite(the clickOn).member
  if the mouseLine < 1 then exit
  lastLine = 0
  repeat while the stillDown
    ML = the mouseLine
    if ML < 1 then next repeat
    if ML <> lastLine then
      thisField.line[ML].hilite()
      lastLine = ML
    end if
  end repeat
end
```

### See also

`mouseChar`, `mouseItem`, `mouseWord`, `line...of`, `number (lines)`

## mouseLoc

### Syntax

the mouseLoc

### Description

Function; returns the current position of the mouse as a point().

The point location is given as two coordinates, with the horizontal location first, then the vertical location.

### See also

mouseH, mouseV

## mouseMember

### Syntax

the mouseMember

### Description

System property; returns the cast member assigned to the sprite that is under the pointer when the property is called. When the pointer is not over a sprite, it returns the result `VOID`.

The `mouseMember` property replaces `mouseCast`, used in earlier versions of Director.

You can use this function to make a movie perform specific actions when the pointer rolls over a sprite and the sprite uses a certain cast member.

The value of the `mouseMember` property can change frequently. To use this property multiple times in a handler with a consistent value, assign the `mouseMember` value to a local variable when the handler starts and use the variable.

For casts other than cast 1, `mouseMember` returns a value that does not distinguish between the cast member and the cast number. To distinguish the cast member and the cast number, use the expression `member (the mouseMember)`; if the user doesn't click a sprite, however, this expression generates a script error.

### Examples

The following statement checks whether the cast member `Off Limits` is the cast member assigned to the sprite under the pointer and displays an alert if it is. This example shows how you can specify an action based on the cast member assigned to the sprite.

```
if the mouseMember = member "Off Limits" then alert "Stay away from there!"
```

This statement assigns the cast member of the sprite under the pointer to the variable `lastMember`:

```
lastMember = the mouseMember
```

### See also

`member` (sprite property), `memberNum`, `clickLoc`, `mouseChar`, `mouseItem`, `mouseLine`, `mouseWord`, `rollover()`, `number` (cast member property)



# mouseOverButton

## Syntax

`sprite whichFlashSprite.mouseOverButton`  
the `mouseOverButton` of `sprite whichFlashSprite`

## Description

Flash sprite property; indicates whether the mouse pointer is over a button in a Flash movie sprite specified by the *whichFlashSprite* parameter (TRUE), or whether the mouse pointer is outside the bounds of the sprite or the mouse pointer is within the bounds of the sprite but over a nonbutton object, such as the background (FALSE).

This property can be tested but not set.

## Example

This frame script checks to see if the mouse pointer is over a navigation button in the Flash movie in sprite 3. If the mouse pointer is over the button, the script updates a text field with an appropriate message; otherwise, the script clears the message.

```
on enterFrame
  case sprite(3).mouseOverButton of
    TRUE:
      member("Message Line").text = "Click here to go to the next page."
    FALSE:
      member("Message Line").text = " "
  end case
  updateStage
end
```

# on mouseUp (event handler)

## Syntax

`on mouseUp`  
    *statement(s)*  
`end`

## Description

System message and event handler; contains statements that are activated when the mouse button is released.

When the mouse button is released, Lingo searches the following locations, in order, for an `on mouseUp` handler: primary event handler, sprite script, cast member script, frame script, and movie script. Lingo stops searching when it reaches the first location that has an `on mouseUp` handler, unless the handler includes the `pass` command to explicitly pass the `mouseUp` message on to the next location.

To create the same response throughout the movie when the user releases the mouse button, set the `mouseUpScript`.

An `on mouseUp` event handler is a good place to put Lingo that changes the appearance of objects—such as buttons—after they are clicked. You can do this by switching the cast member assigned to the sprite after the sprite is clicked and the mouse button is released.

Where you place an `on mouseUp` handler can affect when it runs, as follows:

- To apply the handler to a specific sprite, put it in a sprite script.
- To apply the handler to a cast member in general, put it in a cast member script.
- To apply the handler to an entire frame, put it in a frame script.
- To apply the handler throughout the entire movie, put it in a movie script.

You can override an `on mouseUp` handler by placing an alternative `on mouseUp` handler in a location that Lingo checks before it gets to the handler you want to override. For example, you can override an `on mouseUp` handler assigned to a cast member by placing an `on mouseUp` handler in a sprite script.

If used in a behavior, this event is passed the sprite script or frame script reference `me`.

#### Example

This handler, assigned to sprite 10, switches the cast member assigned to sprite 10 when the user releases the mouse button after clicking the sprite:

```
on mouseUp
    sprite(10).member = member "Dimmed"
end
```

#### See also

`on mouseDown` (event handler)

## the mouseUp (system property)

#### Syntax

the mouseUp

#### Description

System property; indicates whether the mouse button is released (TRUE) or is being pressed (FALSE).

The Director player for Java doesn't update the `mouseUp` property when Lingo is in a repeat loop.

#### Examples

This handler causes the movie to run as long as the user presses the mouse button. The playhead stops when the user releases the mouse button.

```
on exitFrame me
    if the mouseUp then
        go the frame
    end if
end
```

This statement instructs Lingo to exit the repeat loop or handler it is in when the user releases the mouse button:

```
if the mouseUp then exit
```

#### See also

the `mouseDown` (system property), `mouseH`, `mouseV`, the `mouseUp` (system property)

## on mouseUpOutside

### Syntax

```
on mouseUpOutside me
    statement(s)
end
```

### Description

System message and event handler; sent when the user presses the mouse button on a sprite but releases it (away from) the sprite.

### Example

This statement plays a sound when the user clicks the mouse button over a sprite and then releases it outside the bounding rectangle of the sprite:

```
on mouseUpOutside me
    puppetSound "Professor Long Hair"
end
```

### See also

on mouseEnter, on mouseLeave, on mouseWithin

## mouseUpScript

### Syntax

```
the mouseUpScript
```

### Description

System property; determines the Lingo that is executed when the mouse button is released. The Lingo is written as a string, surrounded by quotation marks, and can be a simple statement or a calling script for a handler.

When the mouse button is released and the `mouseUpScript` property is defined, Lingo executes the instructions specified for the `mouseUpScript` property first. Unless the instructions include the `pass` command so that the `mouseUp` message can be passed on to other objects in the movie, no other `on mouseUp` handlers are executed.

When the instructions you've specified for the `mouseUpScript` property are no longer appropriate, turn them off by using the statement `set the mouseUpScript to empty`.

Setting the `mouseUpScript` property accomplishes the same thing as using the `when mouseUp then` command that appeared in earlier versions of Director.

This property can be tested and set. The default value is `EMPTY`.

### Examples

When this statement is in effect and the movie is paused, the movie always continues whenever the user releases the mouse button:

```
the mouseUpScript = "go to the frame +1"
```

With this statement, when the user releases the mouse button after clicking anywhere on the Stage, the movie beeps:

```
the mouseUpScript = "if the clickOn = 0 then beep"
```

This statement sets `mouseUpScript` to the custom handler *myCustomHandler*. A Lingo custom handler must be enclosed in quotation marks when used with the `mouseUpScript` property.

```
the mouseUpScript = "myCustomHandler"
```

**See also**

`stopEvent`, `mouseDownScript`, `on mouseDown` (event handler), `on mouseUp` (event handler)

## mouseV

**Syntax**

```
the mouseV  
mouseV()
```

**Description**

System property; indicates the vertical position of the mouse cursor, in pixels, from the top of the Stage. The value increases as the cursor moves down and decreases as the cursor moves up.

The `mouseV` property is useful for moving sprites to the vertical position of the mouse cursor and checking whether the cursor is within a region of the Stage. Using the `mouseH` and `mouseV` properties together, you can identify the cursor's exact location.

The Director player for Java doesn't update the `mouseV` property when Lingo is in a repeat loop.

This property can be tested but not set.

**Examples**

This handler moves sprite 1 to the mouse pointer location and updates the Stage when the user clicks the mouse button:

```
on mouseDown  
    sprite(1).locH = the mouseH  
    sprite(1).locV = the mouseV  
    updateStage  
end
```

This statement tests whether the pointer is more than 10 pixels above or below a starting point and, if it is, sets the variable `vFar` to `TRUE`:

```
if abs(the mouseV - startV) > 10 then draggedEnough = TRUE
```

**See also**

`mouseH`, `locH`, `locV`, `mouseLoc`

## on mouseWithin

**Syntax**

```
on mouseWithin  
    statement(s)  
end
```

**Description**

System message and event handler; contains statements that run when the mouse is within the active area of the sprite. The mouse button does not have to be pressed.

If the sprite is a bitmap cast member with the matte ink applied, the active area is the portion of the image that is displayed; otherwise, the sprite's bounding rectangle is the active area.

If used in a behavior, this event is passed the sprite script or frame script reference me.

### Example

This statement displays the mouse location when the mouse pointer is over a sprite:

```
on mouseWithin
  member("Display").text = string(the mouseH)
end mouseWithin
```

### See also

on mouseEnter, on mouseLeave

## mouseWord

### Syntax

the mouseWord

### Description

System property; contains the number of the word under the pointer when the property is called and when the pointer is over a field sprite. Counting starts from the beginning of the field. When the mouse is not over a field, the result is -1.

The value of the `mouseWord` property can change in a handler or repeat loop. If a handler or repeat loop uses this property multiple times, it's usually a good idea to call the function once and assign its value to a local variable.

### Examples

This statement determines whether the pointer is over a field sprite and changes the content of the field cast member Instructions to "Please point to a word." when it is not:

```
if the mouseWord = -1 then
  member("Instructions").text = "Please point to a word."
else
  member("Instructions").text = "Thank you."
end if
```

This statement assigns the number of the word under the pointer in the specified field to the variable `currentWord`:

```
currentWord = member(the mouseMember).word[the mouseWord]
```

This handler highlights the word under the pointer when the mouse button is pressed:

```
on mouseDown
  thisField = sprite(the clickOn).member
  if the mouseWord < 1 then exit
  lastWord = 0
  repeat while the stillDown
    MW = the mouseWord
    if MW < 1 then next repeat
    if MW <> lastWord then
      thisField.word[MW].hilite()
      lastWord = MW
    end if
  end repeat
end
```

### See also

mouseChar, mouseItem, mouseLine, number (words), word...of

# moveableSprite

## Syntax

`sprite(whichSprite).moveableSprite`  
the moveableSprite of sprite *whichSprite*

## Description

Sprite property; indicates whether a sprite can be moved by the user (TRUE) or not (FALSE).

You can make a sprite moveable by using the Moveable option in the Score. However, to control whether a sprite is moveable and to turn this condition on and off as needed, use Lingo. For example, to let users drag sprites one at a time and then make the sprites unmoveable after they are positioned, turn the `moveableSprite` sprite property on and off at the appropriate times.

**Note:** For more customized control such as snapping back to the origin or animating while dragging, create a behavior to manage the additional functionality.

This property can be tested and set.

## Examples

This handler makes sprites in channel 5 moveable:

```
on spriteMove
    sprite(5).moveableSprite = TRUE
end
```

This statement checks whether a sprite is moveable and, if it is not, displays a message:

```
if sprite(13).moveableSprite = FALSE then
    member("Notice").text = "You can't drag this item by using the mouse."
```

## See also

`mouseLoc`

# move member

## Syntax

`member(whichCastMember).move()`  
`member(whichCastMember).move(member whichLocation)`  
`move member whichCastMember {,member whichLocation}`

## Description

Command; moves the cast member specified by *whichCastMember* to the first empty location in the Cast window (if no parameter is set) or to the location specified by *whichLocation*.

For best results, use this command during authoring, not at run time, because the move is usually saved with the file. The actual location of a cast member does not affect most presentations during playback for an end user. To switch the content of a sprite or change the display during run time, set the member of the sprite.

## Examples

This statement moves cast member Shrine to the first empty location in the Cast window:

```
member("shrine").move()
```

This statement moves cast member Shrine to location 20 in the Bitmaps Cast window:

```
member("Shrine").move(20, "Bitmaps")
```

## moveToBack

### Syntax

```
window("whichWindow ").MoveToBack()  
moveToBack window whichWindow
```

### Description

Command; moves the window specified by *whichWindow* behind all other windows.

### Examples

These statements move the first window in `windowList` behind all other windows:

```
myWindow = the windowList[1]  
moveToBack myWindow
```

If you know the name of the window you want to move, use the syntax:

```
window("Demo Window").moveToBack()
```

## moveToFront

### Syntax

```
window("whichWindow ").MoveToFront()  
moveToFront window whichWindow
```

### Description

Command; moves the window specified by *whichWindow* in front of all other windows.

### Examples

These statements move the first window in `windowList` in front of all other windows:

```
myWindow = the windowList[1]  
moveToFront myWindow
```

If the you know the name of the window you want to move to the front, use the syntax:

```
window("Demo Window").moveToFront()
```

## moveVertex()

### Syntax

```
member(memberRef). MoveVertex(vertexIndex, xChange, yChange)  
moveVertex(member memberRef, vertexIndex, xChange, yChange)
```

### Description

Function; moves the vertex of a vector shape cast member to another location.

The horizontal and vertical coordinates for the move are relative to the current position of the vertex point. The location of the vertex point is relative to the origin of the vector shape member.

Changing the location of a vertex affects the shape in the same way as dragging the vertex in an editor.

### Example

This statement shifts the first vertex point in the vector shape Archie 25 pixels to the right and 10 pixels down from its current position:

```
member("Archie").moveVertex(1, 25, 10)
```

### See also

`addVertex`, `deleteVertex()`, `moveVertexHandle()`, `originMode`, `vertexList`

## moveVertexHandle()

### Syntax

```
moveVertexHandle(member memberRef, vertexIndex, handleIndex, xChange, yChange)
```

### Description

Function; moves the vertex handle of a vector shape cast member to another location.

The horizontal and vertical coordinates for the move are relative to the current position of the vertex handle. The location of the vertex handle is relative to the vertex point it controls.

Changing the location of a control handle affects the shape in the same way as dragging the vertex in the editor.

### Example

This statement shifts the first control handle of the second vertex point in the vector shape Archie 15 pixels to the right and 5 pixels up:

```
MoveVertexHandle(member "Archie", 2, 1, 15, -5)
```

### See also

`addVertex`, `deleteVertex()`, `originMode`, `vertexList`

## on moveWindow

### Syntax

```
on moveWindow  
    statement(s)  
end
```

### Description

System message and event handler; contains statements that run when a window is moved, such as by dragging a movie to a new location on the Stage, and is a good place to put Lingo that you want executed every time a movie's window changes location.

### Example

This handler displays a message in the Message window when the window a movie is playing in moves:

```
on moveWindow  
    put "Just moved window containing"&&the movieName  
end
```

### See also

`activeWindow`, `movieName`, `windowList`

## movie

This property is obsolete. Use `movieName`.



## movieAboutInfo

### Syntax

the movieAboutInfo

### Description

Movie property; a string entered during authoring in the Movie Properties dialog box. This property is provided to allow for enhancements in future versions of Shockwave.

This property can be set but not tested.

### See also

movieCopyrightInfo

## movieCopyrightInfo

### Syntax

the movieCopyrightInfo

### Description

Movie property; enters a string during authoring in the Movie Properties dialog box. This property is provided to allow for enhancements in future versions of Shockwave.

This property can be tested but not set.

### Example

This statement displays the copyright information in a text cast member:

```
member("Display").text = "Copyright"&&the movieCopyrightInfo
```

### See also

movieAboutInfo

## movieFileFreeSize

### Syntax

the movieFileFreeSize

### Description

Movie property; returns the number of unused bytes in the current movie caused by changes to the cast members and castLibs within a movie. The **Save and Compact** and **Save As** commands rewrite the file to delete this free space.

When the movie has no unused space, the `movieFileFreeSize` function returns 0.

### Example

This statement displays the number of unused bytes that are in the current movie:

```
put the movieFileFreeSize
```

## movieFileSize

### Syntax

`the movieFileSize`

### Description

Movie property; returns the number of bytes in the current movie saved on disk. This is the same number returned when selecting File Properties in Windows or Get Info in the Macintosh Finder.

### Example

This statement displays the number of bytes in the current movie:

```
put the movieFileSize
```

## movieFileVersion

### Syntax

`the movieFileVersion`

### Description

Movie property; indicates the version of Director in which the movie was last saved. The result is a string.

### Example

This statement displays the version of Director that last saved the current movie:

```
put the movieFileVersion  
-- "800"
```

## movieImageCompression

### Syntax

`the movieImageCompression`

### Description

Movie property; indicates the type of compression that Director applies to internal (non-linked) bitmap members when saving a movie in Shockwave format. This property can be set only during authoring and has no affect until the movie is saved in Shockwave format. Its value can be either of these symbols:

Value	Meaning
#standard	Use the Director standard internal compression format
#jpeg	Use JPEG compression (see <code>imageCompression</code> )

You normally set this property in the Director Publish Settings dialog box.

You can choose to override this setting for specific bitmap cast members by setting their `imageCompression` and `imageQuality` cast member properties.

### See also

`imageCompression`, `imageQuality`, `movieImageQuality`

## movieImageQuality

### Syntax

the movieImageQuality

### Description

Movie property; indicates the level of compression to use when the movieImageCompression property is set to #jpeg. The range of acceptable values is 0–100. Zero yields the lowest image quality and highest compression; 100 yields the highest image quality and lowest compression.

You can only set this property during authoring and it has no affect until the movie is saved in Shockwave format.

Individual members may override this movie property by using the member property imageCompression.

### See also

imageCompression, imageQuality, movieImageCompression

## movieName

### Syntax

the movieName

### Description

Movie property; indicates the simple name of the current movie.

In the Director authoring environment, a new movie that has not been saved has an empty string as this property.

### Example

This statement displays the name of the current movie in the Message window:

```
put the movieName
```

### See also

moviePath

## moviePath

### Syntax

the moviePath

### Description

Movie property; indicates the pathname of the folder in which the current movie is located.

For pathnames that work on both Windows and Macintosh computers, use the @ pathname operator.

To see an example of moviePath used in a completed movie, see the Read and Write Text movie in the Learning/Lingo Examples folder inside the Director application folder.

### Examples

This statement displays the pathname for the folder containing the current movie:

```
put the moviePath
```

This statement plays the sound file `Crash.aif` stored in the Sounds subfolder of the current movie's folder. Note the path delimiter used, indicating a Windows environment:

```
sound playFile 1, the moviePath&"Sounds/crash.aif"
```

**See also**

@ (pathname), movieName

## movieRate

**Syntax**

```
sprite(whichSprite).movieRate  
the movieRate of sprite whichSprite
```

**Description**

Digital video sprite property; controls the rate at which a digital video in a specific channel plays. The movie rate is a value specifying the playback of the digital video. A value of 1 specifies normal forward play, -1 specifies reverse, and 0 specifies stop. Higher and lower values are possible. For example, a value of 0.5 causes the digital video to play slower than normal. However, frames may be dropped when the `movieRate` sprite property exceeds 1. The severity of frame dropping depends on factors such as the performance of the computer the movie is playing on and whether the digital video sprite is stretched.

This property can be tested and set.

To see an example of `movieRate` used in a completed movie, see the QT and Flash movie in the Learning/Lingo Examples folder inside the Director application folder.

**Examples**

This statement sets the rate for a digital video in sprite channel 9 to normal playback speed:

```
sprite(9).movieRate = 1
```

This statement causes the digital video in sprite channel 9 to play in reverse:

```
sprite(9).movieRate = -1
```

**See also**

duration, movieTime

## movieTime

**Syntax**

```
sprite(whichSprite).movieTime  
the movieTime of sprite whichSprite
```

**Description**

Digital video sprite property; determines the current time of a digital video movie playing in the channel specified by *whichSprite*. The `movieTime` value is measured in ticks.

This property can be tested and set.

To see an example of `movieTime` used in a completed movie, see the QT and Flash movie in the Learning/Lingo Examples folder inside the Director application folder.

### Examples

This statement displays the current time of the QuickTime movie in channel 9 in the Message window:

```
put sprite(9).movieTime
```

This statement sets the current time of the QuickTime movie in channel 9 to the value in the variable `Poster`:

```
sprite(9).movieTime = Poster
```

### See also

`duration`

## movieXtraList

### Syntax

```
the movieXtraList
```

### Description

Movie property; displays a linear property list of all Xtra extensions in the Movies/Xtras dialog box that have been added to the movie.

- `#name`—Specifies the filename of the Xtra on the current platform. It is possible to have a list without a `#name` entry, such as when the Xtra exists only on one platform.
- `#packagefiles`—Set only when the Xtra is marked for downloading. The value of this property is another list containing a property list for each file in the download package for the current platform. The properties in this subproperty list are `#name` and `#version`, which contain the same information as found in the `XtraList`.

Two possible properties can appear in `movieXtraList`:

### Example

This statement displays output from `movieXtraList` in the Message window:

```
put the movieXtraList
-- [[/#name: "Mix Services"], [/#name: "Sound Import Export"], [/#name: "SWA
  Streaming \ PPC Xtra"], [/#name: "TextXtra PPC"], [/#name: "Font Xtra
  PPC"], [/#name: "Flash Asset \ Options PPC"], [/#name: "Font Asset PPC"],
  [/#name: "Flash Asset PPC", \
  #packagefiles: [[/#fileName: "Flash Asset PPC", #version: "1.0.3"]]]]
```

### See also

`xtraList`

## multiply()

### Syntax

```
transform.multiply(transform2)
```

### Description

3D command; applies the positional, rotational, and scaling effects of *transform2* after the original transform.

### Example

This statement applies the positional, rotational, and scaling effects of the model Mars's transform to the transform of the model Pluto. This has a similar effect as making Mars be Pluto's parent for a frame.

```
member("scene").model("Pluto").transform.multiply(member("scene")\
    .model("Mars").transform)
```

## multiSound

### Syntax

```
the multiSound
```

### Description

System property; specifies whether the system supports more than one sound channel and requires a Windows computer to have a multichannel sound card (TRUE) or not (FALSE).

### Example

This statement plays the sound file Music in sound channel 2 if the computer supports more than one sound channel:

```
if the multiSound then sound playFile 2, "Music.wav"
```

## name

### Syntax

```
member(whichCastmember).texture(whichTexture).name  
member(whichCastmember).shader(whichShader).name  
member(whichCastmember).motion(whichMotion).name  
member(whichCastmember).modelResource(whichModelResource).name  
member(whichCastmember).model(whichModel).name  
member(whichCastmember).light(whichLight).name  
member(whichCastmember).camera(whichCamera).name  
member(whichCastmember).group(whichGroup).name  
node.name
```

### Description

3D property; when used with an object reference, allows you to get the name of the referenced object. You can only get the name; the name can't be changed.

All names must be unique. If created through Lingo, the name returned is the name given in the constructor function. If created through a 3D-authoring program the name returned may be the name of the model.

### Example

This statement sets the name of the fifth camera in the cast member TableScene to BirdCam:

```
member("TableScene").camera[5].name = "BirdCam"
```

## name (cast property)

### Syntax

`castLib (whichCast).name`  
the name of castLib *whichCast*

### Description

Cast member property; returns the name of the specified cast.

This property can be tested and set.

### Example

This code iterates through all the castLibs in a movie and displays their names in the Message window:

```
totalCastLibs = the number of castLibs
repeat with currentCastLib = 1 to totalCastLibs
    put "CastLib"&&currentCastLib&&"is named"&&castLib(currentCastLib).name
end repeat
```

### See also

`&&` (concatenator)

## name (cast member property)

### Syntax

`member(whichCastMember).name`  
the name of member *whichCastMember*

### Description

Cast member property; determines the name of the specified cast member. The argument *whichCastMember* is a string when is used as the cast member name or an integer when used as the cast member number.

The name is a descriptive string assigned by the developer. Setting this property is equivalent to entering a name in the Cast Member Properties dialog box.

This property can be tested and set.

### Examples

This statement changes the name of the cast member named On to Off:

```
member("On").name = "Off"
```

This statement sets the name of cast member 15 to Background Sound:

```
member(15).name = "Background Sound"
```

This statement sets the variable `itsName` to the name of the cast member that follows the cast member whose number is equal to the variable `i`:

```
itsName = member(i + 1).name
```

### See also

`number` (cast member property)

## name (menu property)

### Syntax

the name of menu(*whichMenu*)  
the name of menu *whichMenu*

### Description

Menu property; returns a string containing the name of the specified menu number.

This property can be tested but not set. Use the `installMenu` command to set up a custom menu bar.

**Note:** Menus are not available in Shockwave.

### Examples

This statement assigns the name of menu number 1 to the variable `firstMenu`:

```
firstMenu = menu(1).name
```

The following handler returns a list of menu names, one per line:

```
on menuList  
  theList = []  
  repeat with i = 1 to the number of menus  
    theList[i] = the name of menu i  
  end repeat  
  return theList  
end menuList
```

### See also

`number (menus)`, `name (menu item property)`

## name (menu item property)

### Syntax

the name of menuItem(*whichItem*) of menu(*whichMenu*)  
the name of menuItem *whichItem* of menu *whichMenu*

### Description

Menu property; determines the text that appears in the menu item specified by *whichItem* in the menu specified by *whichMenu*. The *whichItem* argument is either a menu item name or a menu item number; *whichMenu* is either a menu name or a menu number.

This property can be tested and set.

**Note:** Menus are not available in Shockwave.

### Examples

This statement sets the variable `itemName` to the name of the eighth item in the Edit menu:

```
set itemName = the name of menuItem(8) of menu("Edit")
```

This statement causes a specific filename to follow the word *Open* in the File menu:

```
the name of menuItem("Open") of menu("fileMenu") = "Open" && fileName
```

### See also

`name (menu property)`, `number (menu items)`



## name (window property)

### Syntax

`window (whichWindow).name`  
the name of window *whichWindow*

### Description

Window property; determines the name of the specified window in `windowList`. (The title window property determines the title that appears in a window's title bar.)

This property can be tested and set.

### Example

This statement changes the name of the window Yesterday to Today:

```
window("Yesterday").name = "Today"
```

## name (system property)

### Syntax

`xtra (whichXtra).name`  
the name of xtra *whichXtra*

### Description

System property; indicates the name of the specified Lingo Xtra. Xtra extensions that provide support services or other functions not available to Lingo will not support this property.

This property can be tested but not set.

### Example

This statement displays the name of the first Xtra in the Message window:

```
put xtra(1).name
```

## name (timeout property)

### Syntax

`timeoutObject.name`

### Description

This timeout property is the name of the timeout object as defined when the object is created. The `new()` command is used to create timeout objects.

### Example

This timeout handler opens an alert with the name of the timeout that sent the event:

```
on handleTimeout timeoutObject  
    alert "Timeout:" && timeoutObject.name  
end
```

### See also

`forget`, `new()`, `period`, `persistent`, `target`, `time` (timeout object property), `timeout()`, `timeoutHandler`, `timeoutList`

## name (XML property)

### Syntax

*XMLnode.name*

### Description

XML property; returns the name of the specified XML node.

### Example

Beginning with this XML:

```
<?xml version="1.0"?>
  <e1>
    <tagName attr1="val1" attr2="val2"/>
    <e2>element 2</e2>
    <e3>element 3</e3>
  </e1>
```

This Lingo returns the name of the second tag that is nested within the tag <e1>:

```
put gParserObject.child[1].child[2].name
-- "e2"
```

### See also

`attributeName`

## NAN

### Description

Lingo return value; indicates that a specified Lingo expression is not a number.

This statement attempts to display the square root of -1, which is not a number, in the Message window:

```
put (-1).sqrt
-- NAN
```

### See also

`INF`

## near (fog)

### Syntax

```
member(whichCastmember).camera(whichCamera).fog.near
cameraReference.fog.near
member(whichCastmember).camera(whichCamera).fog.far
cameraReference.fog.far
```

### Description

3D properties; this property allows you to get or set the distance from the front of the camera to the point where the fogging starts if `fog.enabled` is `TRUE`.

The default value for this property is 0.0.

### Example

The following statement sets the `near` property of the fog of the camera `Defaultview` to 100. If the fog's `enabled` property is set to `TRUE` and its `decayMode` property is set to `#linear`, fog will first appear 100 world units in front of the camera.

```
member("3d world").camera("defaultview").fog.near = 100.0
```

### See also

`fog`, `far (fog)`, `enabled (fog)`, `decayMode`

## nearFiltering

### Syntax

```
member(whichCastmember).texture(whichTexture).nearFiltering  
member(whichCastmember).shader(whichShader).\  
    texture(whichTexture).nearFiltering  
member(whichCastmember).model(whichModel).shader.texture\  
    (whichTexture).nearFiltering  
member(whichCastmember).model(whichModel).shaderList\  
    [shaderListIndex].texture(whichTexture).nearFiltering
```

### Description

3D texture property; allows you to get or set whether bilinear filtering is used when rendering a projected texture map that covers more screen space than the original texture source. Bilinear filtering smooths any errors across the texture and thus improves the texture's appearance. Bilinear filtering smooths errors in two dimensions. Trilinear filtering smooths errors in three dimensions. Filtering improves appearance at the expense of performance, with bilinear being less performance-costly than trilinear.

When the property's value is `TRUE`, bilinear filtering is used. When the value is `FALSE`, bilinear filtering is not used. The default is `TRUE`.

### Example

This statement turns off bilinear filtering for the texture named `gbTexture` in the cast member `Scene`:

```
member("Scene").texture("gbTexture").nearFiltering = FALSE
```

## neighbor

### Syntax

```
member(whichCastmember).model(whichModel).meshdeform.mesh[index].\  
    face[index].neighbor[index]
```

### Description

3D command; `meshDeform` command that returns a list of lists describing the neighbors of a particular face of a mesh opposite the face corner specified by the neighbor index (1, 2, 3). If the list is empty, the face has no neighbors in that direction. If the list contains more than one list, the mesh is non-manifold. Usually the list contains a single list of four integer values: [`meshIndex`, `faceIndex`, `vertexIndex`, `flipped`].

The value `meshIndex` is the index of the mesh containing the neighbor face. The value `faceIndex` is the index of the neighbor face in that mesh. The value `vertexIndex` is the index of the nonshared vertices of the neighbor face. The value `flipped` describes whether the face orientation is the same as (1) or opposite (2) that of the original face.

### See also

`meshDeform (modifier)`

## netAbort

### Syntax

```
netAbort(URL)  
netAbort(netID)
```

### Description

Command; cancels a network operation without waiting for a result.

Using a network ID is the most efficient way to stop a network operation. The ID is returned when you use a network function such as `getNetText()` or `postNetText()`.

In some cases, when a network ID is not available, you can use a URL to stop the transmission of data for that URL. The URL must be identical to that used to begin the network operation. If the data transmission is complete, this command has no effect.

### Example

This statement passes a network ID to `netAbort` to cancel a particular network operation:

```
on mouseUp  
    netAbort(myNetID)  
end
```

### See also

`getNetText()`, `postNetText`

## netDone()

### Syntax

```
netDone()  
netDone(netID)
```

### Description

Function; indicates whether a background loading operation (such as `getNetText`, `preloadNetThing`, `gotoNetMovie`, `gotoNetPage`, or `putNetText`) is finished or was terminated by a browser error (TRUE, default) or is still in progress (FALSE).

- Use `netDone()` to test the last network operation.
- Use `netDone(netID)` to test the network operation identified by *netID*.

The `netDone` function returns 0 when a background loading operation is in progress.

### Examples

The following handler uses the `netDone` function to test whether the last network operation has finished. If the operation is finished, text returned by `netTextResult` is displayed in the field cast member `Display Text`.

```
on exitFrame  
    if netDone() = 1 then  
        member("Display Text").text = netTextResult()  
    end if  
end
```

This handler uses a specific network ID as an argument for `netDone` to check the status of a specific network operation:

```
on exitFrame
  -- stay on this frame until the net operation is
  -- completed
  global mynetID
  if netDone(mynetID) = FALSE then
    go to the frame
  end if
end
```

#### See also

`getNetText()`, `netTextResult()`, `gotoNetMovie`, `preloadNetThing()`

## netError()

### Syntax

```
netError()
netError(netID)
```

### Description

Function; determines whether an error has occurred in a network operation and, if so, returns an error number corresponding to an error message. If the operation was successful, this function returns a code indicating that everything is okay. If no background loading operation has started, or if the operation is in progress, this function returns an empty string.

- Use `netError()` to test the last network operation.
- Use `netError(netID)` to test the network operation specified by *netID*.

Several possible error codes may be returned:

---

0	Everything is okay.
4	Bad MOA class. The required network or nonnetwork Xtra extensions are improperly installed or not installed at all.
5	Bad MOA Interface. See 4.
6	Bad URL or Bad MOA class. The required network or nonnetwork Xtra extensions are improperly installed or not installed at all.
20	Internal error. Returned by <code>netError()</code> in the Netscape browser if the browser detected a network or internal error.
4146	Connection could not be established with the remote host.
4149	Data supplied by the server was in an unexpected format.
4150	Unexpected early closing of connection.
4154	Operation could not be completed due to timeout.
4155	Not enough memory available to complete the transaction.
4156	Protocol reply to request indicates an error in the reply.
4157	Transaction failed to be authenticated.
4159	Invalid URL.
4164	Could not create a socket.

---

4165	Requested object could not be found (URL may be incorrect).
4166	Generic proxy failure.
4167	Transfer was intentionally interrupted by client.
4242	Download stopped by <code>netAbort(url)</code> .
4836	Download stopped for an unknown reason, possibly a network error, or the download was abandoned.

---

When a movie plays back as an applet, this function always returns a string. The string either has a length of 0 or consists of text that describes an error. The string's content comes from Java and can vary on different operating systems or browsers. The text may not be translated into the local language.

### Example

This statement passes a network ID to `netError` to check the error status of a particular network operation:

```
on exitFrame
  global mynetID
  if netError(mynetID)<>"OK" then beep
end
```

## netLastModDate()

### Syntax

```
netLastModDate()
```

### Description

Function; returns the date last modified from the HTTP header for the specified item. The string is in Universal Time (GMT) format: *Ddd, nn Mmm yyyy hh:mm:ss GMT* (for example, Thu, 30 Jan 1997 12:00:00 AM GMT). There are variations where days or months are spelled completely. The string is always in English.

The `netLastModDate` function can be called only after `netDone` and `netError` report that the operation is complete and successful. After the next operation starts, the Director movie or projector discards the results of the previous operation to conserve memory.

The actual date string is pulled directly from the HTTP header in the form provided by the server. However, this string is not always provided, and in that case `netLastModDate` returns `EMPTY`.

When a movie plays back as an applet, this function's date format may differ from the date format that Shockwave uses; the date is in the format that the Java function `Date.toString` returns. The format may also vary on systems that use different languages.

### Example

These statements check the date of a file downloaded from the Internet:

```
if netDone() then
  theDate = netLastModDate()
  if theDate.char[6..11] <> "Jan 30" then
    alert "The file is outdated."
  end if
end if
```

### See also

```
netDone(), netError()
```

## netMIME()

### Syntax

netMIME()

### Description

Function; provides the MIME type of the Internet file that the last network operation returned (the most recently downloaded HTTP or FTP item).

The netMIME function can be called only after netDone and netError report that the operation is complete and successful. After the next operation starts, the Director movie or projector discards the results of the previous operation to conserve memory.

### Example

This handler checks the MIME type of an item downloaded from the Internet and responds accordingly:

```
on checkNetOperation theURL
  if netDone (theURL) then
    set myMimeType = netMIME()
    case myMimeType of
      "image/jpeg": go frame "jpeg info"
      "image/gif": go frame "gif info"
      "application/x-director": goToNetMovie theURL
      "text/html": goToNetPage theURL
      otherwise: alert "Please choose a different item."
    end case
  else
    go the frame
  end if
end
```

### See also

netDone(), netError(), getNetText(), postNetText, preloadNetThing()

## netPresent

### Syntax

netPresent()  
the netPresent

### Description

System property; determines whether the Xtra extensions needed to access the Internet are available but does not report whether an Internet connection is currently active.

If the Net Support Xtra extensions are not available, netPresent will function properly, but netPresent() will cause a script error

### Example

This statement sends an alert if the Xtra extensions are not available:

```
if not (the netPresent) then
  alert "Sorry, the Network Support Xtras could not be found."
end if
```

## netStatus

### Syntax

```
netStatus msgString
```

### Description

Command; displays the specified string in the status area of the browser window.

The `netStatus` command doesn't work in projectors.

### Example

This statement would place the string "This is a test" in the status area of the browser the movie is running in:

```
on exitFrame
    netStatus "This is a test"
end
```

## netTextResult()

### Syntax

```
netTextResult(netID)
netTextResult()
```

### Description

Function; returns the text obtained by the specified network operation. If no net ID is specified, `netTextResult` returns the result of the last network operation.

If the specified network operation was `getNetText()`, the text is the text of the file on the network.

If the specified network operation was `postNetText`, the result is the server's response.

After the next operation starts, Director discards the results of the previous operation to conserve memory.

When a movie plays back as an applet, this function returns valid results for the last 10 requests. When a movie plays back as a Shockwave movie, this function returns valid results for only the most recent `getNetText()` operation.

### Example

This handler uses the "netDone and netError" functions to test whether the last network operation finished successfully. If the operation is finished, text returned by `netTextResult` is displayed in the field cast member `Display Text`.

```
global gNetID

on exitFrame
    if (netDone(gNetID) = TRUE) and (netError(gNetID) = "OK") then
        member("Display Text").text = netTextResult()
    end if
end
```

### See also

`netDone()`, `netError()`, `postNetText`



## netThrottleTicks

### Syntax

the netThrottleTicks

### Description

System property; in the Macintosh authoring environment, allows you to control the frequency of servicing to a network operation.

The default value is 15. The higher the value is set, the smoother the movie playback and animation is, but less time is spent servicing any network activity. A low setting allows more time to be spent on network operations, but will adversely affect playback and animation performance.

This property only affects the authoring environment and projectors on the Macintosh. It is ignored on Windows or Shockwave on the Mac.

## new()

### Syntax

```
new(type)
new(type, castLib whichCast)
new(type, member whichCastMember of castLib whichCast)
variableName = new(parentScript arg1, arg2, ...)
new(script parentScriptName, value1, value2, ...)
timeout("name").new(timeoutPeriod, #timeoutHandler, {, targetObject})
new(xtra "xtraName")
```

### Description

Function; creates a new cast member, child object, timeout object, or Xtra instance and allows you to assign of individual property values to child objects.

The Director player for Java supports this function only for the creation of child objects. When a movie plays back as an applet, you can't use the `new` function to create cast members.

For cast members, the `type` parameter sets the cast member's type. Possible predefined values correspond to the existing cast member types: `#bitmap`, `#field`, and so on. The `new` function can also create Xtra cast member types, which can be identified by any name that the author chooses.

It's also possible to create a new color cursor cast member using the Custom Cursor Xtra. Use `new(#cursor)` and set the properties of the resulting cast member to make them available for use.

The optional `whichCastMember` and `whichCast` parameters specify the cast member slot and Cast window where the new cast member is stored. When no cast member slot is specified, the first empty slot is used. The `new` function returns the cast member slot.

When the argument for the `new` function is a parent script, the `new` function creates a child object. The parent script should include an `on new` handler that sets the child object's initial state or property values and returns the `me` reference to the child object.

The child object has all the handlers of the parent script. The child object also has the same property variable names that are declared in the parent script, but each child object has its own values for these properties.

Because a child object is a value, it can be assigned to variables, placed in lists, and passed as a parameter.

As with other variables, you can use the `put` command to display information about a child object in the Message window.

When `new()` is used to create a timeout object, the `timeoutPeriod` sets the number of milliseconds between timeout events sent by the timeout object. The `#timeoutHandler` is a symbol that identifies the handler that will be called when each timeout event occurs. The `targetObject` identifies the name of the child object that contains the `#timeoutHandler`. If no `targetObject` is given, the `#timeoutHandler` is assumed to be in a movie script.

When a timeout object is created, it enables its `targetObject` to receive the system events `prepareMovie`, `startMovie`, `stopMovie`, `prepareFrame`, and `exitFrame`. To take advantage of this, the `targetObject` must contain handlers for these events. The events do not need to be passed in order for the rest of the movie to have access to them.

To see an example of `new()` used in a completed movie, see the Parent Scripts, and Read and Write Text movies in the Learning/Lingo Examples folder inside the Director application folder.

### Examples

To create a new bitmap cast member in the first available slot, you use this syntax:

```
set newMember = new(#bitmap)
```

After the line has been executed, `newMember` will contain the member reference to the cast member just created:

```
put newMember  
-- (member 1 of castLib 1)
```

The following `startMovie` script creates a new Flash cast member using the `new` command, sets the newly created cast member's `linked` property so that the cast member's assets are stored in an external file, and then sets the cast member's `pathName` property to the location of a Flash movie on the World Wide Web:

```
on startMovie  
  flashCastMember = new(#flash)  
  member(flashCastMember).pathName = "http://www.someURL.com/myFlash.swf"  
end
```

When the movie starts, this handler creates a new animated color cursor cast member and stores its cast member number in a variable called `customCursor`. This variable is used to set the `castMemberList` property of the newly created cursor and to switch to the new cursor.

```
on startMovie  
  customCursor = new(#cursor)  
  member(customCursor).castMemberList = [member 1, member 2, member 3]  
  cursor (member(customCursor))  
end
```

These statements from a parent script include the `on new` handler to create a child object. The parent script is a script cast member named `Bird`, which contains these handlers.

```
on new me, nameForBird  
  return me  
end  
  
on fly me  
  put "I am flying"  
end
```

The first statement in the following example creates a child object from the above script in the preceding example, and places it in a variable named `myBird`. The second statement makes the bird fly by calling the fly handler in the Bird parent script:

```
myBird = script("Bird").new()  
myBird.fly()
```

This statement uses a new Bird parent script, which contains the property variable `speed`:

property `speed`

```
on new me, initSpeed  
    speed = initSpeed  
    return me  
end  
on fly me  
    put "I am flying at " & speed & "mph"  
end
```

The following statements create two child objects called `myBird1` and `myBird2`. They are given different starting speeds: 15 and 25, respectively. When the fly handler is called for each child object, the speed of the object is displayed in the Message window.

```
myBird1 = script("Bird").new(15)  
myBird2 = script("Bird").new(25)  
myBird1.fly()  
myBird2.fly()
```

This message appears in the Message window:

```
-- "I am flying at 15 mph"  
-- "I am flying at 25 mph"
```

This statement creates a new timeout object called `intervalTimer` that will send a timeout event to the `on minuteBeep` handler in the child object `playerOne` every 60 seconds:

```
timeout("intervalTimer").new(60000, #minuteBeep, playerOne)
```

#### See also

`on stepFrame`, `actorList`, `ancestor`, `me`, `type` (cast member property), `timeout()`

## newCamera

### Syntax

```
member(whichCastmember).newCamera(newCameraName)
```

### Description

3D command; creates a new camera, *newCameraName*, within the cast member. The name specified for *newCameraName* must be unique within the cast member.

### Example

This statement creates a new camera called in-car camera:

```
member("3D World").newCamera("in-car camera")
```

## newCurve()

### Syntax

```
vectorMember.newCurve(positionInVertexList)  
newCurve(vectorMember, positionInVertexList)
```

### Description

Function; adds a *#newCurve* symbol to the *vertexList* of *vectorCastMember*, which adds a new shape to the vector shape. The *#newCurve* symbol is added at *positionInVertexList*. You can break apart an existing shape by calling *newCurve()* with a position in the middle of a series of vertices.

### Example

This Lingo adds a new curve to cast member 2 at the third position in the cast member's *vertexList*. The second line of the example replaces the contents of curve 2 with the contents of curve 3.

```
member(2).newCurve(3)  
member(2).curve[2]=member(2).curve[3]  
curve, vertexList
```

## newGroup

### Syntax

```
member(whichCastmember).newGroup(newGroupName)
```

### Description

3D command; creates a new group, *newGroupName*, and adds it to the group palette. You cannot have two groups in the palette with the same name.

### Example

This statement creates a group called *gbGroup2* within the cast member *Scene*, and a reference to it is stored in the variable *ng*:

```
ng = member("Scene").newGroup("gbGroup2")
```

## newLight

### Syntax

```
member(whichCastmember).newLight(newLightName, #typeIndicator)
```

### Description

3D command; creates a new light named, *newLightName*, with the type *#typeIndicator*, and adds it to the light palette. You can not have two lights in the palette with the same name.

The *#typeIndicator* parameter has the following possible values:

- *#ambient* is a generalized light in the 3D world.
- *#directional* is a light from a specific direction.
- *#point* is a light source like a light bulb.
- *#spot* is a spotlight effect.

### Example

The following statement creates a new light in the cast member named *3D World*. It is an ambient light called "ambient room light".

```
member("3D World").newLight("ambient room light", #ambient)
```

# newMesh

## Syntax

`member(whichCastmember).newMesh(name, numFaces, numVertices, numNormals, numColors, numTextureCoordinates)`

## Description

3D command; creates a new mesh model resource using the arguments supplied. Note that after creating a mesh, you must set values for at least the `vertexList` and `face[index].vertices` properties of the new mesh, followed by a call to its `build()` command, in order to actually generate the geometry.

The parameters of `newMesh` are as follows:

- `numFaces` is the desired total number of triangles you want in the mesh.
- `numVertices` is the total number of vertices used by all the (triangular) faces. A vertex may be shared by more than one face.
- `numNormals` is the optional total number of normals. A normal may be shared by more than one face. The normal for a corner of a triangle defines which direction is outward, affecting how that corner is illuminated by lights. Enter 0 or omit this argument if you are going to use the mesh's `generateNormals()` command to generate normals.
- `numColors` is the optional total number of colors used by all the faces. A color may be shared by more than one face. You can specify a color for each corner of each face. Specify colors for smooth color gradation effects. Enter 0 or omit this argument to get default white color per face corner.
- `numTextureCoordinates` is the optional number of user-specified texture coordinates used by all the faces. Enter 0 or omit this argument to get the default texture coordinates generated via a planar mapping. (See the explanation of `#planar` in the `shader.textureWrapMode` entry for more details). Specify texture coordinates when you need precise control over how textures are mapped onto the faces of the mesh.

## Example

This example creates a model resource of the type `#mesh`, specifies its properties, and then creates a new model from the model resource. The process is outlined in the following line-by-line explanation of the example code:

**Line 1** creates a mesh containing 6 faces, composed of 5 unique vertices and 3 unique colors. The number of normals and the number of `textureCoordinates` are not set. The normals will be created by the `generateNormals` command.

**Line 2** defines the five unique vertices used by the faces of the mesh.

**Line 3** defines the three unique colors used by the faces of the mesh.

**Lines 4 through 9** assign which vertices to use as the corners of each face in the Pyramid. Note the clockwise ordering of the vertices. `GenerateNormals()` relies on a clockwise ordering.

**Lines 10 through 15** assign colors to the corners of each face. The colors will spread across the faces in gradients.

**Line 16** creates the normals of Triangle by calling the `generateNormals()` command.

**Line 17** calls the `build` command to construct the mesh.

```
nm = member("Shapes").newMesh("pyramid",6 , 5, 0, 3)
nm.vertexList = [ vector(0,0,0), vector(40,0,0), \
    vector(40,0,40), vector(0,0,40), vector(20,50,20) ]
nm.colorList = [ rgb(255,0,0), rgb(0,255,0), rgb(0,0,255) ]
nm.face[1].vertices = [ 4,1,2 ]
nm.face[2].vertices = [ 4,2,3 ]
nm.face[3].vertices = [ 5,2,1 ]
nm.face[4].vertices = [ 5,3,2 ]
nm.face[5].vertices = [ 5,4,3 ]
nm.face[6].vertices = [ 5,1,4 ]
nm.face[1].colors = [3,2,3]
nm.face[2].colors = [3,3,2]
nm.face[3].colors = [1,3,2]
nm.face[4].colors = [1,2,3]
nm.face[5].colors = [1,3,2]
nm.face[6].colors = [1,2,3]
nm.generateNormals(#flat)
nm.build()
nm = member("Shapes").newModel("Pyramid1", nm)
```

#### See also

`newModelResource`

## newModel

### Syntax

```
member( whichCastmember ).newModel( newModelName \
    {, whichModelResource } )
```

### Description

3D command; creates a new model in the referenced cast member. The *newModelName* must be unique, as the command fails if a model by that name already exists. All new models have their resource property set to void by default. You can use the optional second parameter to specify a model resource to create the model from.

### Examples

This statement creates a model called New House within the cast member 3D World.

```
member("3D World").newModel("New House")
```

Alternatively, the model resource for the new model can be set with the optional *whichModelResource* parameter.

```
member("3D World").newModel("New House", member("3D \
World").modelResource("bigBox"))
```

# newModelResource

## Syntax

```
member(whichCastmember).newModelResource(newModelResourceName \
    { ,#type, #facing })
```

## Description

3D command; creates a new model resource, of the given *#type* and *#facing* (if specified), and adds it to the model resource palette.

The *#type* parameter can be one of the following primitives:

```
#plane
#box
#sphere
#cylinder
#particle
```

If you do not choose to specify the *#facing* parameter and specify *#box*, *#sphere*, *#particle* or *#cylinder* for the *#type* parameter, only the front faces are generated, if you specify *#plane*, both the front and back faces are generated. Model resources of the type *#plane* have two meshes generated (one for each side), and consequently has two shaders in the *shaderList*.

The *#facing* parameter can be one of the following:

- *#front*
- *#back*
- *#both*

A facing of *#both* creates the double amount of meshes and consequently produces double the number of shader entries in the *shaderList*. There will be 2 for planes and spheres (for the inside and outside of the model respectively), 12 for cubes (6 on the outside, 6 on the inside), and 6 for cylinders (top, hull and bottom outside, and another set for the inside).

## Examples

The following handler creates a box. The first line of the handler creates a new model resource called *box10*. Its type is *#box*, and it is set to show only its back. The next three lines set the dimensions of *box10* and the last line creates a new model which uses *box10* as its model resource.

```
on makeBox
    nmr = member("3D").newModelResource("box10", #box, #back)
    nmr.height = 50
    nmr.width = 50
    nmr.length = 50
    aa = member("3D").newModel("gb5", nmr)
end
```

This statement creates a box-shaped model resource called *hatbox4*.

```
member("Shelf").newModelResource("hatbox4", #box)
```

## See also

primitives

## newMotion()

### Syntax

`member(whichCastmember).newMotion(name)`

### Description

3D command; creates a new motion within the referenced cast member, and returns a reference to the new motion. A new motion can be used to combine several previously existing motions from the member's motion list via the `map()` command. All motions within a referenced cast member must have a unique name.

### Example

This Lingo creates a new motion in member 1 called `runWithWave` that is used to combine the run and wave motions from the member's motion list:

```
runWithWave = member(1).newMotion("runWithWave")
runWithWave.map("run", "pelvisBone")
runWithWave.map("wave", "shoulderBone")
```

## newObject()

### Syntax

`flashSpriteReference.newObject("objectType" {, arg1, arg2 ....})`  
`newObject("objectType" {, arg1, arg2 ....})`

### Description

Flash sprite command; creates an `ActionScript` object of the specified type. Any initialization parameters required by the object can be specified after the object type. Each argument must be separated by a comma. The command returns a reference to the new object.

The following syntax creates an object within a Flash sprite:

```
flashSpriteReference.newObject("objectType" {, arg1, arg2 ....})
```

The following syntax creates a global object:

```
newObject("objectType" {, arg1, arg2 ....})
```

**Note:** If you have not imported any Flash cast members, you must manually add the Flash Asset Xtra to your movie's Xtra list in order for global Flash commands to work correctly in Shockwave and projectors. You add Xtra extensions to the Xtra list by choosing `Modify > Movie > Xtras`. For more information, see "Managing Xtra extensions for distributed movies" in Director Help (`Help > Using Director > Packaging Movies for Distribution`).

### Examples

This Lingo sets the variable `tLocalConObject` to a reference to a new `LocalConnection` object in the Flash movie in sprite 3:

```
tLocalConObject = sprite(3).newObject("LocalConnection")
```

The following Lingo sets the variable `tArrayObject` to a reference to a new array object in the Flash movie in sprite 3. The array contains the 3 integer values 23, 34, and 19.

```
tArrayObject = sprite(3).newObject("Array",23,34,19)
```

### See also

`setCallback()`, `clearAsObjects()`



# newShader

## Syntax

`member(whichCastmember).newShader(newShaderName, #shaderType)`

## Description

3D command; creates a new shader of the specified *#shaderType* within the referenced cast member's shader list and returns a reference to the new shader. All shaders in the shader list must have a unique name. The *#shaderType* argument determines the style in which the shader is applied and the has the following possible values:

- *#standard* shaders are photorealistic, and have the following properties: *ambient*, *blend*, *blendConstant*, *blendConstantList*, *blendFunction*, *blendFunctionList*, *blendSource*, *blendSourceList*, *diffuse*, *diffuseLightMap*, *emissive*, *flat*, *glossMap*, *ilk*, *name*, *region*, *renderStyle*, *silhouettes*, *specular*, *specularLightMap*, *texture*, *textureMode*, *textureModeList*, *textureRepeat*, *textureRepeatList*, *textureTransform*, *textureTransformList*, *transparent*, *useDiffuseWithTexture*, *wrapTransform*, and *wrapTransformList*.
- *#painter* shaders are smoothed out, have the appearance of a painting, and have the following properties in addition to all of the *#standard* properties: *colorSteps*, *hilightPercentage*, *hilightStrength*, *name*, *shadowPercentage*, *shadowStrength*, and *style*.
- *#engraver* shaders are lined, have the appearance of an engraving, and have the following properties in addition to all of the *#standard* properties: *brightness*, *density*, *name*, and *rotation*.
- *#newsprint* shaders are in a simulated dot style, have the appearance of a newspaper reproduction, and have the following properties in addition to all of the *#standard* properties: *brightness*, *density*, and *name*.

Each type of shader has a specific group of properties that can be used with that type of shader, in addition all shader types have access to the *#standard* shader properties. However, although you can assign any *#standard* shader property to a shader of another type, the property may not have a visual effect. This happens in cases where the *#standard* property, if applied, would override the nature of the shader type. An example of this is the *diffuseLightMap* standard shader property, which is ignored by *#engraver*, *#newsprint*, and *#painter* type shaders.

## Example

This statement creates a *#painter* shader called *newPainter*:

```
newPainter = member("3D World").newShader("newPainter",#painter)
```

## See also

*shadowPercentage*

## newTexture

### Syntax

```
member(whichCastmember).newTexture(newTextureName \
    {,#typeIndicator, sourceObjectReference})
```

### Description

3D command; creates a new texture within the referenced member's texture palette and returns a reference to the new texture. All textures in the member's texture palette must have a unique name. The *#typeIndicator* and *sourceObjectReference* parameters are optional, and if not specified a new texture with no type or source is created. The only way cast member textures will work is if you specify the cast member in the newTexture constructor.

The *#typeIndicator* parameter can have two values, *#fromCastMember* (a cast member) or *#fromImageObject* (a lingo image object). The *sourceObjectReference* parameter must be a cast member reference if you specify *#fromCastMember*, or must be a Lingo image object if you specify *#fromImageObject*.

### Example

The first line of this statement creates a new texture called Grass 02 from cast member 5 of castlib 1. The second line creates a blank new texture called Blank.

```
member("3D World").newTexture("Grass \
    02",#fromCastMember,member(5,1))
member("3D World").newTexture("Blank")
```

## next

### Syntax

```
next
```

### Description

Keyword; refers to the next marker in the movie and is equivalent to the phrase the marker (+ 1).

### Examples

This statement sends the playhead to the next marker in the movie:

```
go next
```

This handler moves the movie to the next marker in the Score when the right arrow key is pressed and to the previous marker when the left arrow key is pressed:

```
on keyUp
    if the keyCode = 124 then go next
    if the keyCode = 123 then go previous
end keyUp
```

### See also

```
loop (keyword),go previous
```

## next repeat

### Syntax

`next repeat`

### Description

Keyword; sends Lingo to the next step in a repeat loop in a script. This function differs from that of the `exit repeat` keyword.

### Example

This repeat loop displays only odd numbers in the Message window:

```
repeat with i = 1 to 10
    if (i mod 2) = 0 then next repeat
    put i
end repeat
```

## node

### Syntax

`sprite(whichQTVRSprite).node`  
the node of sprite *whichQTVRSprite*

### Description

QuickTime VR sprite property; the current node ID displayed by the sprite.

This property can be tested and set.

## nodeEnterCallback

### Syntax

`sprite(whichQTVRSprite).nodeEnterCallback`  
the nodeEnterCallback of sprite *whichQTVRSprite*

### Description

QuickTime VR sprite property; contains the name of the handler that runs after the QuickTime VR movie switches to a new active node on the Stage. The message has two arguments: the `me` parameter and the ID of the node that is being displayed.

The QuickTime VR sprite receives the message first.

To clear the callback, set this property to 0.

To avoid a performance penalty, set a callback property only when necessary.

This property can be tested and set.

## nodeExitCallback

### Syntax

```
sprite(whichQTVRSprite).nodeExitCallback  
the nodeExitCallback of sprite whichQTVRSprite
```

### Description

QuickTime VR sprite property; contains the name of the handler that runs when the QuickTime VR movie is about to switch to a new active node on the Stage. The message has three arguments: the `me` parameter, the ID of the node that the movie is about to leave, and the ID of the node that the movie is about to switch to.

The value that the handler returns determines whether the movie goes on to the next node. If the handler returns `#continue`, the QuickTime VR sprite continues with a normal node transition. If the handler returns `#cancel`, the transition doesn't occur and the movie stays in the original node.

Set this property to 0 to clear the callback.

The QuickTime VR sprite receives the message first.

To avoid a performance penalty, set a callback property only when necessary.

This property can be tested and set.

## nodeType

### Syntax

```
sprite(whichQTVRSprite).nodeType  
nodeType of sprite whichQTVRSprite
```

### Description

QuickTime VR sprite property; gives the type of node that is currently on the Stage for the specified sprite. Possible values are `#object`, `#panorama`, or `#unknown`. (`#unknown` is the value for a sprite that isn't a QuickTime VR sprite.)

This property can be tested but not set.

## normalize

### Syntax

```
normalize(vector)  
vector.normalize()
```

### Description

3D command; normalizes a vector by dividing the *x*, *y*, and *z* components by the vector's magnitude. Vectors that have been normalized always have a magnitude of 1.

### Example

This statement shows the value of the vector `MyVec` before and after being normalized:

```
MyVec = vector(-209.9019, 1737.5126, 0.0000)  
MyVec.normalize()  
put MyVec  
-- vector(-0.1199, 0.9928, 0.0000)  
put MyVec.magnitude  
-- 1.0000
```

This statement shows the value of the vector `ThisVector` before and after being normalized.

```
ThisVector = vector(-50.0000, 0.0000, 0.0000)
normalize(ThisVector)
put ThisVector
-- vector(-1.0000, 0.0000, 0.0000)
```

**See also**

`getNormalized`, `randomVector`, `magnitude`

## normalList

**Syntax**

```
member(whichCastmember).modelResource(whichModelResource).\
    normalList
model.meshDeform.mesh[index].normalList
```

**Description**

3D property; when used with a model resource whose type is `#mesh`, this property allows you to get or set the `normalList` property of the model resource.

The `normalList` property is a linear list of vectors from which you may specify vertex normals when building the faces of your mesh.

This property must be set to a list of exactly the number of vectors specified in the `newMesh()` call.

Alternately, the `normalList` property may be generated for you by the `generateNormals()` method of mesh model resources.

In the context of the `meshDeform` modifier, the `normalList` property is similarly a linear list of vectors from which you may specify vertex normals when deforming your mesh.

For more information on face normals and vertex normals, see the `normals` entry.

**Examples**

```
put member(5,2).modelResource("mesh square").normalList
-- [vector(0,0,1)]
member(2).modelResource("mesh3").normalList[2] = vector\
    (205.0000, -300.0000, 27.0000)
```

**See also**

`face`, `meshDeform` (modifier)

## normals

**Syntax**

```
member(whichCastmember).modelResource(whichModelResource).\
    face[index].normals
```

**Syntax**

3D face property; for model resources whose type is `#mesh` (created using the `newMesh` command) this property allows you to get and set the list of normal vectors used by the face specified by the `index` parameter.

Set this property to a linear list of integers corresponding to the index position of each vertex's normal in the model resource's `normalList` property.

This property must be set to the same length as the `face[index].vertices` list, or it can be an empty list `[]`.

Do not set any value for this property if you are going to generate normal vectors using the `generateNormals()` command.

If you make changes to this property, or use the `generateNormals()` command, you need to call the `build()` command in order to rebuild the mesh.

#### Example

This statement sets the `normals` property of the fifth face of the model resource named `Player` to a list of integer values:

```
member("3D").modelResource("Player").face[5].normals = [2,32,14]
```

#### See also

`face`, `normalList`, `vertices`

## not

#### Syntax

```
not logicalExpression
```

#### Description

Operator; performs a logical negation on a logical expression. This is the equivalent of making a TRUE value FALSE, and making a FALSE value TRUE. It is useful when testing to see if a certain known condition is not the case.

This logical operator has a precedence level of 5.

#### Examples

This statement determines whether 1 is not less than 2:

```
put not (1 < 2)
```

Because 1 is less than 2, the result is 0, which indicates that the expression is FALSE.

This statement determines whether 1 is not greater than 2:

```
put not (1 > 2)
```

Because 1 is not greater than 2, the result is 1, which indicates that the expression is TRUE.

This handler sets the `checkMark` menu item property for `Bold` in the `Style` menu to the opposite of its current setting:

```
on resetMenuItem  
  the checkMark of menuItem("Bold") of menu("Style") = \  
    not (the checkMark of menuItem("Bold") of menu("Style"))  
end resetMenuItem
```

#### See also

`and`, `or`

## nothing

### Syntax

nothing

### Description

Command; does nothing. This command is useful for making the logic of an `if...then` statement more obvious. A nested `if...then...else` statement that contains no explicit command for the `else` clause may require `else nothing`, so that Lingo does not interpret the `else` clause as part of the preceding `if` clause.

### Examples

The nested `if...then...else` statement in this handler uses the `nothing` command to satisfy the statement's `else` clause:

```
on mouseDown
  if the clickOn = 1 then
    if sprite(1).moveableSprite = TRUE then
      member("Notice").text = "Drag the ball"
    else nothing
    else member("Notice").text = "Click again"
  end if
end
```

This handler instructs the movie to do nothing so long as the mouse button is being pressed:

```
on mouseDown
  repeat while the stillDown
    nothing
  end repeat
end mouseDown
```

### See also

`if`

## nudge

### Syntax

```
sprite(whichQTVRSprite).nudge(#direction )
nudge(sprite whichQTVRSprite, #direction)
```

### Description

QuickTime VR command; nudges the view perspective of the specified QuickTime VR sprite in the direction specified by *#direction*. Possible values for *#direction* are `#down`, `#downLeft`, `#downRight`, `#left`, `#right`, `#up`, `#upLeft`, and `#upRight`. Nudging to the right causes the image of the sprite to move to the left.

The `nudge` command has no return value.

### Example

This handler causes the perspective of the QTVR sprite to move to the left as long as the mouse button is held down on the sprite:

```
on mouseDown me
  repeat while the stillDown
    sprite(1).nudge(#left)
  end repeat
end
```

## number (cast property)

### Syntax

the number of castLib *whichCast*

### Description

Cast property; indicates the number of the specified cast. For example, 2 is the castLib number for Cast 2.

This property can be tested but not set.

### Example

This repeat loop uses the Message window to display the number of cast members that are in each of the movie's casts:

```
repeat with n = 1 to the number of castLibs
  put castLib(n).name && "contains" && the number of \
  members of castLib(n) && "cast members."
end repeat
```

## number (cast member property)

### Syntax

member(*whichCastMember*).number  
the number of member *whichCastMember*

### Description

Cast member property; indicates the cast number of the cast member specified by *whichCastMember*: either a name, if *whichCastMember* is a string, or a number, if *whichCastMember* is an integer.

The property is a unique identifier for the cast member that is a single integer describing its location in and position in the castLib.

This property can be tested but not set.

**Note:** When using the first syntax of member(*whichCastMember*).number, an error is generated if the cast member does not exist. When unsure of the existence of the member, use the alternate syntax to avoid the error.

### Examples

This statement assigns the cast number of the cast member Power Switch to the variable *whichCastMember*:

```
whichCastMember = member("Power Switch").number
```

This statement assigns the cast member Red Balloon to sprite 1:

```
sprite(1).member = member("Red Balloon").number
```

This verifies that a cast member actually exists before trying to switch the cast member in the sprite:

```
property spriteNum
```

```
on mouseUp me
  if (member("Mike's face").number > 0) then
    sprite(spriteNum).member = "Mike's face"
  end if
end
```

### See also

member (sprite property), memberNum, number of members



## number (characters)

### Syntax

the number of chars in *chunkExpression*

### Description

Chunk expression; returns a count of the characters in a chunk expression.

Chunk expressions are any character (including spaces and control characters such as tabs and carriage returns), word, item, or line in any container of characters. Containers include field cast members and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

**Note:** The `count()` function provides a more efficient alternative for determining the number of characters in a chunk expression.

### Examples

This statement displays the number of characters in the string “Macromedia, the Multimedia Company” in the Message window:

```
put the number of chars in "Macromedia, the Multimedia Company"
```

The result is 34.

This statement sets the variable `charCounter` to the number of characters in the word `i` located in the string `Names`:

```
charCounter = the number of chars in member("Names").word[i]
```

You can accomplish the same thing with text cast members using the syntax:

```
charCounter = member("Names").word[i].char.count
```

### See also

`length()`, `char...of`, `count()`, `number (items)`, `number (lines)`, `number (words)`

## number (items)

### Syntax

the number of items in *chunkExpression*

### Description

Chunk expression; returns a count of the items in a chunk expression. An item chunk is any sequence of characters delimited by commas.

Chunk expressions are any character, word, item, or line in any container of characters. Containers include fields (field cast members) and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

**Note:** The `count()` function provides a more efficient alternative for determining the number of items in a chunk expression.

### Examples

This statement displays the number of items in the string “Macromedia, the Multimedia Company” in the Message window:

```
put the number of items in "Macromedia, the Multimedia Company"
```

The result is 2.

This statement sets the variable `itemCounter` to the number of items in the field `Names`:

```
itemCounter = the number of items in member("Names").text
```

You can accomplish the same thing with text cast members using the syntax:

```
itemCounter = member("Names").item.count
```

**See also**

`item...of`, `count()`, `number (characters)`, `number (lines)`, `number (words)`

## number (lines)

**Syntax**

the number of lines in *chunkExpression*

**Description**

Chunk expression; returns a count of the lines in a chunk expression. (Lines refers to lines delimited by carriage returns, not lines formed by line wrapping.)

Chunk expressions are any character, word, item, or line in any container of characters. Containers include field cast members and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

**Note:** The `count()` function provides a more efficient alternative for determining the number of lines in a chunk expression.

**Examples**

This statement displays the number of lines in the string “Macromedia, the Multimedia Company” in the Message window:

```
put the number of lines in "Macromedia, the Multimedia Company"
```

The result is 1.

This statement sets the variable `lineCounter` to the number of lines in the field `Names`:

```
lineCounter = the number of lines in member("Names").text
```

You can accomplish the same thing with text cast members with the syntax:

```
lineCounter = member("Names").line.count
```

**See also**

`line...of`, `count()`, `number (characters)`, `number (items)`, `number (words)`

## number (menus)

**Syntax**

the number of menus

**Description**

Menu property; indicates the number of menus installed in the current movie.

This menu property can be tested but not set. Use the `installMenu` command to set up a custom menu bar.

**Note:** Menus are not available in Shockwave

### Examples

This statement determines whether any custom menus are installed in the movie and, if no menus are already installed, installs the menu *Menubar*:

```
if the number of menus = 0 then installMenu "Menubar"
```

This statement displays in the Message window the number of menus that are in the current movie:

```
put the number of menus
```

### See also

```
installMenu, number (menu items)
```

## number (menu items)

### Syntax

```
the number of menuItems of menu whichMenu
```

### Description

Menu property; indicates the number of menu items in the custom menu specified by *whichMenu*. The *whichMenu* parameter can be a menu name or menu number.

This menu property can be tested but not set. Use the *installMenu* command to set up a custom menu bar.

**Note:** Menus are not available in Shockwave.

### Examples

This statement sets the variable *fileItems* to the number of menu items in the custom File menu:

```
fileItems = the number of menuItems of menu "File"
```

This statement sets the variable *itemCount* to the number of menu items in the custom menu whose menu number is equal to the variable *i*:

```
itemCount = the number of menuItems of menu i
```

### See also

```
installMenu, number (menus)
```

## number (system property)

### Syntax

```
the number of castLibs
```

### Description

System property; returns the number of casts that are in the current movie.

This property can be tested but not set.

### Example

This repeat loop uses the Message window to display the number of cast members that are in each of the movie's casts:

```
repeat with n = 1 to the number of castLibs
  put castLib(n).name && "contains" && the number of \
    members of castLib(n) && "cast members."
end repeat
```

## number (words)

### Syntax

the number of words in *chunkExpression*

### Description

Chunk expression; returns the number of words in the chunk expression specified by *chunkExpression*.

Chunk expressions are any character, word, item, or line in any container of characters. Containers include field cast members and variables that hold strings, and specified characters, words, items, lines, and ranges in containers.

To accomplish this functionality with text cast members, see `count`.

**Note:** The `count()` function provides a more efficient alternative for determining the number of words in a chunk expression.

### Examples

This statement displays in the Message window the number of words in the string “Macromedia, the multimedia company”:

```
put the number of words in "Macromedia, the multimedia company"
```

The result is 4.

This handler reverses the order of words in the string specified by the argument `wordList`:

```
on reverse wordList
  theList = EMPTY
  repeat with i = 1 to the number of words in wordList
    put word i of wordList & " " before theList
  end repeat
  delete theList.char[theList.char.count]
  return theList
end
```

### See also

`count()`, `number (characters)`, `number (items)`, `number (lines)`, `word...of`

## number of members

### Syntax

the number of members of *castLib whichCast*

### Description

Cast member property; indicates the number of the last cast member in the specified cast.

This property can be tested but not set.

### Example

The following statement displays in the Message window the type of each cast member in the cast Central Casting. The number of members of `castLib` property is used to determine how many times the loop repeats.

```
repeat with i = 1 to the number of members of castLib("Central Casting")
  put "Cast member" && i && "is a" && member(i, "Central Casting").type
end repeat
```

## number of xtras

### Syntax

the number of xtras

### Description

System property; returns the number of scripting Xtra extensions available to the movie. The Xtra extensions may be either those opened by the `openxlib` command or those present in the standard Xtras folder.

This property can be tested but not set.

### Example

This statement displays in the Message window the number of scripting Xtra extensions that are available to the movie:

```
put the number of xtras
```

## numChannels

### Syntax

`member(whichCastMember).numChannels`  
the numChannels of member *whichCastMember*

### Description

Shockwave Audio (SWA) cast member property; returns the number of channels within the specified SWA streaming cast member. The value can be either 1 for monaural or 2 for stereo.

This property is available only after the SWA streaming cast member begins playing or after the file has been preloaded using the `preLoadBuffer` command.

This property can be tested but not set.

### Example

This example assigns the number of sound channels of the SWA streaming cast member Duke Ellington to the field cast member Channel Display:

```
myVariable = member("Duke Ellington").numChannels
if myVariable = 1 then
    member("Channel Display").text = "Mono"
else
    member("Channel Display").text = "Stereo"
end if
```

## numParticles

### Syntax

`member(whichCastmember).modelResource(whichModelResource).\  
emitter.numParticles`  
`modelResourceObjectReference.emitter.numParticles`

### Description

3D property; when used with a model resource whose type is `#particle`, allows you to get or set the `numParticles` property of the resource's particle emitter. The value must be greater than 0 and no more than 100000. The default setting is 1000.

### Example

In this example, ThermoSystem is a model resource of the type #particle. This statement sets the number of particles in ThermoSystem to 50000.

```
member("Fires").modelResource("ThermoSystem").emitter.\
  numParticles = 50000
```

### See also

emitter

## numSegments

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
  numSegments
```

### Description

3D property; when used with a model resource whose type is #cylinder, allows you to get or set the numSegments property of the model resource.

The numSegments property determines the number of segments running from the top cap of the cylinder to the bottom cap. This property must be greater than or equal to the default value of 2.

The smoothness of the cylinder's surface depends upon the value specified for this property. The greater the property value the smoother the cylinder's surface will appear.

### Example

This statement sets the numSegments property of the model resource named Cylinder03 to 10:

```
member("3D World").modelResource("Cylinder03").numSegments = 10
```

## numToChar()

### Syntax

```
numToChar(integerExpression)
```

### Description

Function; displays a string containing the single character whose ASCII number is the value of *integerExpression*. This function is useful for interpreting data from outside sources that are presented as numbers rather than as characters.

ASCII values up to 127 are standard on all computers. Values of 128 or greater refer to different characters on different computers.

### Examples

This statement displays in the Message window the character whose ASCII number is 65:

```
put numToChar(65)
```

The result is the letter *A*.

This handler removes any nonalphabetic characters from any arbitrary string and returns only capital letters:

```
on ForceUppercase input
  output = EMPTY
  num = length(input)
  repeat with i = 1 to num
    theASCII = charToNum(input.char[i])
    if theASCII = min(max(96, theASCII), 123) then
      theASCII = theASCII - 32
    if theASCII = min(max(63, theASCII), 91) then
      put numToChar(theASCII) after output
    end if
  end repeat
  return output
end
```

**See also**

charToNum()

## obeyScoreRotation

**Syntax**

```
member(flashMember).obeyScoreRotation
```

**Description**

Flash cast member property; set to `TRUE` or `FALSE` to determine if a Flash movie sprite uses the rotation information from the Score, or the older rotation property of Flash assets.

This property is automatically set to `FALSE` for all movies created in Director prior to version 7 in order to preserve old functionality of using the member rotation property for all sprites containing that Flash member.

New assets created in version 7 or later will have this property automatically set to `TRUE`.

If set to `TRUE`, the rotation property of the member is ignored and the Score rotation settings are obeyed instead.

**Example**

The following sprite script sets the `obeyScoreRotation` property of cast member "dalmation" to 1 (`TRUE`), then rotates the sprite which contains the cast member 180°.

```
on mouseUp me
  member("dalmation").obeyScoreRotation = 1
  sprite(1).rotation = sprite(1).rotation + 180
end
```

This property can be tested and set.

**See also**

rotation

## objectP()

### Syntax

`objectP(expression)`

### Description

Function; indicates whether the expression specified by *expression* is an object produced by a parent script, Xtra, or window (TRUE) or not (FALSE).

The *P* in `objectP` stands for *predicate*.

It is good practice to use `objectP` to determine which items are already in use when you create objects by parent scripts or Xtra instances.

To see an example of `objectP()` used in a completed movie, see the Read and Write Text movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This Lingo checks whether the global variable `gDataBase` has an object assigned to it and, if not, assigns one. This check is commonly used when you perform initializations at the beginning of a movie or section that you don't want to repeat.

```
if objectP(gDataBase) then
    nothing
else
    gDataBase = script("Database Controller").new()
end if
```

### See also

`floatP()`, `ilk()`, `integerP()`, `stringP()`, `symbolP()`

## of

The word `of` is part of many Lingo properties, such as `foreColor`, `number`, `name`, and so on.

## offset() (string function)

### Syntax

`offset(stringExpression1, stringExpression2)`

### Description

Function; returns an integer indicating the position of the first character of *stringExpression1* in *stringExpression2*. This function returns 0 if *stringExpression1* is not found in *stringExpression2*. Lingo counts spaces as characters in both strings.

On the Macintosh, the string comparison is not sensitive to case or diacritical marks. For example, Lingo considers *a* and *Á* to be the same character on the Macintosh.

### Examples

This statement displays in the Message window the beginning position of the string “media” within the string “Macromedia”:

```
put offset("media","Macromedia")
```

The result is 6.



This statement displays in the Message window the beginning position of the string “Micro” within the string “Macromedia”:

```
put offset("Micro", "Macromedia")
```

The result is 0, because “Macromedia” doesn’t contain the string “Micro”.

This handler finds all instances of the string represented by `stringToFind` within the string represented by `input` and replaces them with the string represented by `stringToInsert`:

```
on SearchAndReplace input, stringToFind, stringToInsert
  output = ""
  findLen = stringToFind.length - 1
  repeat while input contains stringToFind
    currOffset = offset(stringToFind, input)
    output = output & input.char [1..currOffset]
    delete the last char of output
    output = output & stringToInsert
    delete input.char [1.. (currOffset + findLen)]
  end repeat
  set output = output & input
  return output
end
```

#### See also

`chars()`, `length()`, `contains`, `starts`

## offset() (rectangle function)

### Syntax

```
rectangle.offset(horizontalChange, verticalChange)
offset (rectangle, horizontalChange, verticalChange)
```

### Description

Function; yields a rectangle that is offset from the rectangle specified by *rectangle*. The horizontal offset is the value specified by *horizontalChange*; the vertical offset is the value specified by *verticalChange*.

- When *horizontalChange* is greater than 0, the offset is toward the right of the Stage; when *horizontalChange* is less than 0, the offset is toward the left of the Stage.
- When *verticalChange* is greater than 0, the offset is toward the top of the Stage; when *verticalChange* is less than 0, the offset is toward the bottom of the Stage.

The values for *verticalChange* and *horizontalChange* are in pixels.

### Example

This handler moves sprite 1 five pixels to the right and five pixels down:

```
on diagonalMove
  newRect=sprite(1).rect.offset(5, 5)
  sprite(1).rect=newRect
end
```

## on

### Syntax

```
on handlerName {argument1}, {arg2}, {arg3} ...  
    statement(s)  
end handlerName
```

### Description

**Keyword;** indicates the beginning of a handler, a collection of Lingo statements that you can execute using the handler name. A handler can accept arguments as input values and returns a value as a function result.

Handlers can be defined in behaviors, movie scripts, and cast member scripts. A handler in a cast member script can be called only by other handlers in the same script. A handler in a movie script can be called from anywhere.

You can use the same handler in more than one movie by putting the handler's script in a shared cast.

## open

### Syntax

```
open {whichDocument with} whichApplication
```

### Description

**Command;** launches the application specified by the string *whichApplication*. Use *whichDocument* to specify a document that the application opens when it is launched. When either is in a different folder than the current movie, you must specify the full pathname to the file or files.

The computer must have enough memory to run both Director and other applications at the same time.

This is a very simple command for opening an application or a document within an application. For more control, look at options available in third-party Xtra extensions.

### Examples

This statement checks whether the computer is a Macintosh and then if it is, opens the application TextEdit:

```
if the platform contains "Mac" then open "TextEdit"
```

This statement opens the TextEdit application, which is in the folder Applications on the drive HD, and the document named Storyboards:

```
open "Storyboards" with "HD:Applications:TextEdit"
```

### See also

`openXlib`

## openResFile

This is obsolete. Use `recordFont`.

## open window

### Syntax

```
window(whichWindow).open()  
open window whichWindow
```

### Description

Window command; opens the window object or movie file specified by *whichWindow* and brings it to the front of the Stage. If no movie is assigned to the window, the Open File dialog box appears.

- If you replace *whichWindow* with a movie's filename, the window uses the filename as the window.
- If you replace *whichWindow* with a window name, the window takes that name. However, you must then assign a movie to the window by using `set the fileName of window`.

To open a window that uses a movie from a URL, it's a good idea to use the `downloadNetThing` command to download the movie's file to a local disk first and then use the file on the disk. This minimizes problems with waiting for the movie to download.

For local media, the movie is not loaded into memory until the `open movie` command is executed. This can create a noticeable delay if you don't use `preloadMovie` to load at least the first frame of the movie prior to issuing the `open window` command.

**Note:** Opening a movie in a window is currently not supported in playback using a browser.

### Example

This statement opens the window Control Panel and brings it to the front:

```
window("Control Panel").open()
```

### See also

`close window`, `downloadNetThing`, `preloadMovie`

## on openWindow

### Syntax

```
on openWindow  
    statement(s)  
end
```

### Description

System message and event handler; contains statements that run when Director opens the movie as a movie in a window and is a good place to put Lingo that you want executed every time the movie opens in a window.

### Example

This handler plays the sound file Hurray when the window that the movie is playing in opens:

```
on openWindow  
    puppetSound 2, "Hurray"  
end
```

## openXlib

### Syntax

`openXlib whichFile`

### Description

Command; opens the Xlibrary file specified by the string expression *whichFile*. If the file is not in the folder containing the current movie, *whichFile* must include the pathname.

It is good practice to close any file you have opened as soon as you are finished using it. The `openXlib` command has no effect on an open file.

The `openXlib` command doesn't support URLs as file references.

Xlibrary files contain Xtra extensions. Unlike `openResFile`, `openXlib` makes these Xtra extensions known to Director.

In Windows, the `.dll` extension is optional.

**Note:** This command is not supported in Shockwave.

### Examples

This statement opens the Xlibrary file Video Disc Xlibrary:

```
openXlib "Video Disc Xlibrary"
```

This statement opens the Xlibrary file Xtras, which is in a different folder than the current movie:

```
openXlib "My Drive:New Stuff:Transporter Xtras"
```

### See also

`closeXlib`, `interface()`, `showXlib`

## optionDown

### Syntax

`the optionDown`

### Description

System property; determines whether the user is pressing the Alt key (Windows) or the Option key (Macintosh) (TRUE) or not (FALSE).

In Windows, `optionDown` doesn't work in projectors if Alt is pressed without another nonmodifier key. Avoid using `optionDown` if you intend to distribute a movie as a Windows projector and need to detect only the modifier key press; use `controlDown` or `shiftDown` instead.

On the Macintosh, pressing the Option key changes the `key` value, so use `keyCode` instead.

For a movie playing back with the Director player for Java, this function returns TRUE only if a second key is pressed at the same time as the Alt or Option key. If the Alt or Option key is pressed by itself, `optionDown` returns FALSE.

The Director player for Java supports key combinations with the Alt or Option key. However, the browser receives the keys before the movie plays and responds to and intercepts any key combinations that are also browser keyboard shortcuts.

**Example**

This handler checks whether the user is pressing the Alt or the Option key and, if so, calls the handler named `doOptionKey`:

```
on keyDown
  if (the optionDown) then doOptionKey(key)
end keyDown
```

**See also**

`controlDown`, `commandDown`, `key()`, `keyCode()`, `shiftDown`

## or

**Syntax**

*logicalExpression1 or logicalExpression2*

**Description**

Operator; performs a logical OR operation on two or more logical expressions to determine whether any expression is `TRUE`.

This is a logical operator with a precedence level of 4.

**Examples**

This statement indicates in the Message window whether at least one of the expressions `1 < 2` and `1 > 2` is `TRUE`:

```
put (1 < 2) or (1 > 2)
```

Because the first expression is `TRUE`, the result is 1, which is the numerical equivalent of `TRUE`.

This Lingo checks whether the content of the field cast member named `State` is either `AK` or `HI` and displays an alert if it is:

```
if member("State").text = "AK" or member("State").text = "HI" then
  alert "You're off the map!"
end if
```

**See also**

`and`, `not`

## organizationName

**Syntax**

`the organizationName`

**Description**

Movie property; contains the company name entered during installation of Director.

This property is available in the authoring environment only. It can be used in a movie in a window tool that is personalized to show the user's information.

### Example

The following handler would be located in a movie script of a movie in a window (MIAW). It places the user's name and serial number into a display field when the window is opened:

```
on prepareMovie
    displayString = the userName
    put RETURN & the organizationName after displayString
    put RETURN & the serialNumber after displayString
    member("User Info").text = displayString
end
```

### See also

serialNumber, userName, window

## originalFont

### Syntax

```
member(whichFontMember).originalFont  
the originalFont of member whichFontMember
```

### Description

Font cast member property; returns the exact name of the original font that was imported when the given cast member was created.

### Example

This statement displays the name of the font that was imported when cast member 11 was created:

```
put member(11).originalFont  
-- "Monaco"
```

### See also

recordFont, bitmapSizes, characterSet

## originH

### Syntax

```
sprite(whichVectorOrFlashSprite).originH  
the originH of sprite whichVectorOrFlashSprite  
member(whichVectorOrFlashMember).originH  
the originH of member whichVectorOrFlashMember
```

### Description

Cast member and sprite property; controls the horizontal coordinate of a Flash movie or vector shape's origin point, in pixels. The value can be a floating-point value.

The origin point is the coordinate in a Flash movie or vector shape around which scaling and rotation occurs. The origin point can be set with floating-point precision using the separate `originH` and `originV` properties, or it can be set with integer precision using the single `originPoint` property.

You can set the `originH` property only if the `originMode` property is set to `#point`.

This property can be tested and set. The default value is 0.

**Note:** This property must be set to the default value if the `scaleMode` property is set to `#autoSize`, or the sprite does not display correctly.

### Example

This sprite script uses the `originMode` property to set up a Flash movie sprite so its origin point can be set to a specific point. It then sets the horizontal and vertical origin points.

```
on beginSprite me
    sprite(spriteNum of me).originMode = #point
    sprite(spriteNum of me).originH = 100
    sprite(spriteNum of me).originV = 80
end
```

### See also

`originV`, `originMode`, `originPoint`, `scaleMode`

## originMode

### Syntax

`sprite(whichFlashOrVectorShapeSprite).originMode`  
the `originMode` of sprite *whichFlashOrVectorShapeSprite*  
`member(whichFlashOrVectorShapeMember).originMode`  
the `originMode` of member *whichFlashOrVectorShapeMember*

### Description

Cast member property and sprite property; sets the origin point around which scaling and rotation occurs, as follows:

- `#center` (default)—The origin point is at the center of the Flash movie.
- `#topleft`—The origin point is at the top left of the Flash movie.
- `#point`—The origin point is at a point specified by the `originPoint`, `originH`, and `originV` properties.

This property can be tested and set.

**Note:** This property must be set to the default value if the `scaleMode` property is set to `#autoSize`, or the sprite will not display correctly.

### Example

This sprite script uses the `originMode` property to set up a Flash movie sprite so its origin point can be set to a specific point. It then sets the horizontal and vertical origin points.

```
on beginSprite me
    sprite(spriteNum of me).originMode = #point
    sprite(spriteNum of me).originH = 100
    sprite(spriteNum of me).originV = 80
end
```

### See also

`originH`, `originV`, `originPoint`, `scaleMode`

# originPoint

## Syntax

`sprite whichVectorOrFlashSprite.originPoint`  
the `originPoint` of `sprite whichVectorOrFlashSprite`  
`member(whichVectorOrFlashMember).originPoint`  
the `originPoint` of `member whichVectorOrFlashMember`

## Description

Cast member and sprite property; controls the origin point around which scaling and rotation occurs of a Flash movie or vector shape.

The `originPoint` property is specified as a Director point value; for example, `point(100,200)`. Setting a Flash movie or vector shape's origin point with the `originPoint` property is the same as setting the `originH` and `originV` properties separately. For example, setting the `originPoint` property to `point(50,75)` is the same as setting the `originH` property to 50 and the `originV` property to 75.

Director point values specified for the `originPoint` property are restricted to integers, whereas `originH` and `originV` can be specified with floating-point numbers. When you test the `originPoint` property, the point values are truncated to integers. As a rule of thumb, use the `originH` and `originV` properties for precision; use the `originPoint` property for speed and convenience.

You can set the `originPoint` property only if the `originMode` property is set to `#point`.

This property can be tested and set. The default value is 0.

**Note:** This property must be set to the default value if the `scaleMode` property is set to `#autoSize`, or the sprite will not display correctly.

## Example

This sprite script uses the `originMode` property to set up a Flash movie sprite so its origin point can be set to a specific point. It then sets the origin points.

```
on beginSprite me
  sprite(me.spriteNum).scaleMode = #showAll
  sprite(me.spriteNum).originMode = #point
  sprite(me.spriteNum).originPoint = point(100, 80)
end
```

## See also

`originH`, `originV`, `scaleMode`

# originV

## Syntax

`sprite(whichVectorOrFlashSprite).originV`  
the `originV` of `sprite whichVectorOrFlashSprite`  
`member(whichVectorOrFlashMember).originV`  
the `originV` of `member whichVectorOrFlashMember`

## Description

Cast member and sprite property; controls the vertical coordinate of a Flash movie or vector shape's origin point around which scaling and rotation occurs, in pixels. The value can be a floating-point value.



The origin point can be set with floating-point precision using the separate `originH` and `originV` properties, or it can be set with integer precision using the single `originPoint` property.

You can set the `originV` property only if the `originMode` property is set to `#point`.

This property can be tested and set. The default value is 0.

**Note:** This property must be set to the default value if the `scaleMode` property is set to `#autoSize`, or the sprite does not display correctly.

### Example

This sprite script uses the `originMode` property to set up a Flash movie sprite so its origin point can be set to a specific point. It then sets the horizontal and vertical origin points.

```
on beginSprite me
    sprite(me.spriteNum).scaleMode = #showAll
    sprite(me.spriteNum).originMode = #point
    sprite(me.spriteNum).originH = 100
    sprite(me.spriteNum).originV = 80
end
```

### See also

`originH`, `originPoint`, `scaleMode`

## otherwise

### Syntax

```
otherwise statement(s)
```

### Description

Keyword; precedes instructions that Lingo performs when none of the earlier conditions in a case statement are met.

This keyword can be used to alert users of out-of-bound input or invalid type, and can be very helpful in debugging during development.

### Example

The following handler tests which key the user pressed most recently and responds accordingly:

- If the user pressed A, B, or C, the movie performs the corresponding action following the of keyword.
- If the user pressed any other key, the movie executes the statement that follows the `otherwise` keyword. In this case, the statement is a simple alert.

```
on keyDown
    case (the key) of
        "A": go to frame "Apple"
        "B", "C":
            puppetTransition 99
            go to frame "Oranges"
        otherwise:
            alert "That is not a valid key."
    end case
end keyDown
```

## orthoHeight

### Syntax

```
member(whichCastmember).camera(whichCamera).orthoHeight  
member(whichCastmember).camera[cameraIndex].orthoHeight  
sprite(whichSprite).camera.orthoHeight
```

### Description

3D property; when `camera.projection` is set to `#orthographic`, the value `camera.orthoHeight` gives the number of perpendicular world units that fit vertically in the sprite. World units are the measuring units for the particular 3D world. They are internally consistent but arbitrarily chosen, and they can vary from one 3D world to another.

Note that you don't need to specify the camera index (*whichCamera*) to access the first camera of the sprite.

The default value of this property is 200.0

### Example

The following statement sets the `orthoHeight` of the camera of sprite 5 to 200. This means 200 world units will fit vertically within the sprite.

```
sprite(5).camera.orthoheight = 200.0
```

### See also

`projection`

## overlay

### Syntax

```
member(whichCastmember).camera(whichCamera).\  
    overlay[overlayIndex].propertyName  
member(whichCastmember).camera(whichCamera).overlay.count
```

### Description

3D camera property; allows both get and set access to the properties of overlays contained in the camera's list of overlays to be displayed. When used as `overlay.count` this property returns the total number of overlays contained in the camera's list of overlays to be displayed.

Overlays are textures displayed in front of all models appearing in a given camera's view frustum. The overlays are drawn in the order that they appear in the camera's overlay list, the first item in the list appears behind all other overlays and the last item in the list in front of all other overlays.

Each overlay in the camera's list of overlays list has the following properties:

- `loc` allows you to get or set the specific position of the overlay's `regPoint`, relative to the camera `rect`'s upper left corner.
- `source` allows you to get or set the texture to use as the source image for the overlay.
- `scale` allows you to get or set the scale value used by the overlay. The scale determines the magnification of the overlay; this property defaults to a value of 1.0.
- `rotation` allows you to get or set the rotation, in degrees, of the overlay.
- `regPoint` allows you to get or set the registration point of the overlay relative to the texture's upper left corner.

- `blend` allows you to get or set the blending of the overlay to an integer between 0 and 100, indicating how transparent (0) or opaque (100) the overlay is.

#### Example

This statement displays the scale property of the third overlay in the sprite camera's overlay list:

```
put sprite(5).camera.overlay[3].scale
-- 0.5000
```

#### See also

`addOverlay`, `removeOverlay`, `bevelDepth`

## pageHeight

#### Syntax

`member(whichCastMember).pageHeight`  
the `pageHeight` of member *whichCastMember*

#### Description

Field cast member property; returns the height, in pixels, of the area of the field cast member that is visible on the Stage.

This property can be tested but not set.

#### Example

This statement returns the height of the visible portion of the field cast member Today's News:

```
put member("Today's News").pageHeight"
```

## palette

#### Syntax

`member(whichCastMember).palette`  
the palette of member *whichCastMember*

#### Description

Cast member property; for bitmap cast members only, determines which palette is associated with the cast member specified by *whichCastMember*.

This property can be tested and set.

#### Example

This statement displays the palette assigned to the cast member Leaves in the Message window:

```
put member("Leaves").palette"
```

## paletteMapping

#### Syntax

the `paletteMapping`

#### Description

Movie property; determines whether the movie remaps (`TRUE`) or does not remap (`FALSE`, default) palettes for cast members whose palettes are different from the current movie palette. Its effect is similar to that of the Remap Palettes When Needed check box in the Movie Properties dialog box.

To display different bitmaps with different palettes simultaneously, set `paletteMapping` to `TRUE`. Director looks at each onscreen cast member's reference palette (the palette assigned in its Cast Member Properties dialog box) and, if it is different from the current palette, finds the closest match for each pixel in the new palette.

The colors of the nonmatching bitmap will be close to the original colors.

Remapping consumes processor time, and it's usually better to adjust the bitmap's palette in advance.

Remapping can also produce undesirable results. If the palette changes in the middle of a sprite span, the bitmap immediately remaps to the new palette and appears in the wrong colors. However, if anything refreshes the screen—a transition or a sprite moving across the Stage—then the affected rectangle on the screen appears in remapped colors.

#### Example

This statement tells the movie to remap the movie's palette whenever necessary:

```
set the paletteMapping = TRUE
```

## paletteRef

#### Syntax

```
member(whichCastMember). paletteRef  
the paletteRef
```

#### Description

Bitmap cast member property; determines the palette associated with a bitmap cast member. Built-in Director palettes are indicated by symbols (`#systemMac`, `#rainbow`, and so on). Palettes that are cast members are treated as cast member references. This behavior differs from that of the `palette` member property, which returns a positive number for cast palettes and negative numbers for built-in Director palettes.

This property can be tested and set.

#### Example

This statement assigns the Macintosh system palette to the bitmap cast member Shell:

```
member("Shell").paletteRef = #systemMac
```

## pan (QTVR property)

#### Syntax

```
pan of sprite whichQTVRSprite
```

#### Description

QuickTime VR sprite property; the current pan of the QuickTime VR movie. The value is in degrees.

This property can be tested and set.

## pan (sound property)

### Syntax

`sound(channelNum).pan`  
the pan of `sound(channelNum)`

### Description

Property; indicates the left/right balance of the sound playing in sound channel *channelNum*. The range of values is from -100 to 100. -100 indicates only the left channel is heard. 100 indicate only the right channel is being heard. A value of 0 indicates even left/right balance, causing the sound source to appear to be centered. For mono sounds, *pan* affects which speaker (left or right) the sound plays through.

You can change the pan of a sound object at any time, but if the sound channel is currently performing a fade, the new pan setting doesn't take effect until the fade is complete.

To see an example of *pan* (sound property) used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This Lingo pans the sound in sound channel 2 from the left channel to the right channel:

```
repeat with x = -100 to 100
    sound(2).pan = x
end repeat
```

### See also

`fadeIn()`, `fadeOut()`, `fadeTo()`, `volume (sound channel)`

## paragraph

### Syntax

`chunkExpression.paragraph[whichParagraph]`  
`chunkExpression.paragraph[firstParagraph..lastParagraph]`

### Description

Text cast member property; this chunk expression allows access to different paragraphs within a text cast member.

The paragraph is delimited by a carriage return.

```
put member("AnimText").paragraph[3]
```

### See also

`line...of`

## param()

### Syntax

`param(parameterPosition)`

### Description

Function; provides the value of a parameter passed to a handler. The expression *parameterPosition* represents the parameter's position in the arguments.

To avoid errors in a handler, this function can be used to determine the type of a particular parameter.

### Example

This handler accepts any number of arguments, adds all the numbers passed in as parameters, and then returns the sum:

```
on AddNumbers
  sum = 0
  repeat with currentParamNum = 1 to the paramCount
    sum = sum + param(currentParamNum)
  end repeat
  return sum
end
```

You would use it by passing in the values you wanted to add:

```
put AddNumbers(3, 4, 5, 6)
-- 18
put AddNumbers(5, 5)
-- 10
```

### See also

getAt, param(), paramCount(), return (keyword)

## paramCount()

### Syntax

the paramCount

### Description

Function; indicates the number of parameters sent to the current handler.

### Example

This statement sets the variable `counter` to the number of parameters that were sent to the current handler:

```
set counter = the paramCount
```

## parent

### Syntax

```
member(whichCastmember).model(whichModel).parent
member(whichCastmember).camera(whichCamera).parent
member(whichCastmember).light(whichLight).parent
member(whichCastmember).group(whichGroup).parent
```

### Description

3D property; when used with a model, camera, light or group reference, this property allows you to get or set the parent node of the referenced object. The parent node can be any other model, camera, light or group object.

An object's `transform` property defines its scale, position and orientation relative to its parent object.

Setting an object's `parent` property to `VOID` is the same as removing the object from the world using the `removeFromWorld()` command.

Setting an object's `parent` property to the World group object (`group("World")`) is the same as adding an object to the world using the `addToWorld()` command.

You can also alter the value of this property by using the `addChild` command.

#### Example

The following statement sets the parent property of the model named Tire. Its parent is set to the model named Car.

```
member("Scene").model("Tire").parent = \
    member("Scene").model("Car")
```

#### See also

`child`, `addChild`

## parseString()

#### Syntax

```
parserObject.parseString(stringToParse)
```

#### Description

Function; used to parse an XML document that is already fully available to the Director movie. The first parameter is the variable containing the parser object, and the second is the string containing the XML data. The return value is `<VOID>` if the operation succeeds, or an error code number string if it fails. Failure is usually due to a problem with the XML syntax or structure. Once the operation is complete, the parser object contains the parsed XML data.

To parse XML at a URL, use `parseURL()`.

#### Example

This statement parses the XML data in the text cast member XMLtext. Once the operation is complete, the variable `gParserObject` will contain the parsed XML data.

```
errorCode = gParserObject.parseString(member("XMLtext"))
```

#### See also

`getError()` (XML), `parseURL()`

## parseURL()

#### Syntax

```
parserObject.parseURL(URLstring {, #handlerToCallOnCompletion} {,
    objectContainingHandler})
```

#### Description

Function; parses an XML document that resides at an external Internet location. The first parameter is the parser object containing an instance of the XML Parser Xtra, and the second is the actual URL at which the XML data resides.

This function returns immediately, so the entire URL may not yet be parsed. It is important to use the `doneParsing()` function in conjunction with `parseURL()` to determine when the parsing operation is complete.

Since this operation is asynchronous, meaning it may take some time, you can use optional parameters to call a specific handler when the operation completes. The optional third parameter is the name of a handler that is to be executed once the URL is fully parsed; the optional fourth parameter is the name of the script object containing that handler. If the fourth parameter is not passed in, the handler name in parameter three is assumed to be a movie handler.

The return value is void if the operation succeeds, or an error code number string if it fails.

To parse XML locally, use `parseString()`.

### Examples

This statement parses the file `sample.xml` at `MyCompany.com`. Use `doneParsing()` to determine when the parsing operation has completed.

```
errorCode = gParserObject.parseURL("http://www.MyCompany.com/sample.xml")
```

This Lingo parses the file `sample.xml` and calls the `on parseDone` handler. Because no script object is given with the `doneParsing()` function, the `on parseDone` handler is assumed to be in a movie script.

```
errorCode = gParserObject.parseURL("http://www.MyCompany.com/sample.xml",  
    #parseDone)
```

The movie script contains the `on parseDone` handler:

```
on parseDone  
    global gParserObject  
    if voidP(gParserObject.getError()) then  
        put "Successful parse"  
    else  
        put "Parse error:"  
        put "    " & gParserObject.getError()  
    end if  
end
```

This Lingo parses the document `sample.xml` at `MyCompany.com` and calls the `on parseDone` handler in the script object `testObject`, which is a child of the parent script `TestScript`:

```
parserObject = new(xtra "XMLParser")  
testObject = new(script "TestScript", parserObject)  
errorCode = gParserObject.parseURL("http://www.MyCompany.com/sample.xml",  
    #parseDone, testObject)
```

Here is the parent script `TestScript`:

```
property myParserObject  
  
on new me, parserObject  
    myParserObject = parserObject  
end  
  
on parseDone me  
    if voidP(myParserObject.getError()) then  
        put "Successful parse"  
    else  
        put "Parse error:"  
        put "    " & myParserObject.getError()  
    end if  
end
```

### See also

`getError()` (XML), `parseString()`



## pass

### Syntax

`pass`

### Description

Command; passes an event message to the next location in the message hierarchy and enables execution of more than one handler for a given event.

The Director player for Java supports this command only within `on keyDown` and `on keyUp` handlers attached to editable sprites.

The `pass` command branches to the next location as soon as the command runs. Any Lingo that follows the `pass` command in the handler does not run.

By default, an event message stops at the first location containing a handler for the event, usually at the sprite level.

If you include the `pass` command in a handler, the event is passed to other objects in the hierarchy even though the handler would otherwise intercept the event.

### Example

This handler checks the keypresses being entered, and allows them to pass through to the editable text sprite if they are valid characters:

```
on keyDown me
    legalCharacters = "1234567890"
    if legalCharacters contains the key then
        pass
    else
        beep
    end if
end
```

### See also

`stopEvent`

## password

### Syntax

```
sprite(whichSprite).password
member(whichCastmember).password
sprite(whichSprite).password = password
member(whichCastmember).password = password
```

### Description

RealMedia sprite and cast member property; allows you to set the password required to access a protected RealMedia stream. For security reasons, you cannot use this property to retrieve a password previously specified for this property. If a password has been set previously, the value of this property is the string "\*\*\*\*\*". If no password has been set, the value of this property is an empty string.

### Examples

The following examples show that the password has been set for the RealMedia stream in the cast member Real or sprite 2.

```
put_sprite(2).password
-- "*****"

put_member("Real").password
-- "*****"
```

The following examples show that the password has never been set for the RealMedia stream in the cast member Real or sprite 2.

```
put_sprite(2).password
-- ""

put_member("Real").password
-- ""
```

The following examples set the password for the RealMedia stream in sprite 2 and the cast member Real to "abracadabra".

```
sprite(2).password = "abracadabra"
member("Real").password = "abracadabra"
```

### See also

`userName (RealMedia)`

## pasteClipboardInto

### Syntax

```
member(whichCastMember). pasteClipboardInto()
pasteClipboardInto member whichCastMember
```

### Description

Command; pastes the contents of the Clipboard into the cast member specified by *whichCastMember* and erases the exiting cast member. For example, pasting a bitmap into a field cast member makes the bitmap the cast member and erases the field cast member.

You can paste any item that is in a format that Director can use as a cast member. When you copy a string from another application, the string's formatting is not retained.

The `pasteClipboardInto` command provides a convenient way to copy objects from other movies and from other applications into the Cast window. Because copied cast members must be stored in RAM, avoid using this command during playback in low memory situations.

**Note:** When you use this command in Shockwave, or in the authoring environment and projectors with the `safePlayer` property set to `TRUE`, a warning dialog will allow the user to cancel the paste operation.

### Example

This statement pastes the Clipboard contents into the bitmap cast member Shrine:

```
member("shrine").pasteClipboardInto()
```

### See also

`safePlayer`

# path

## Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
    emitter.path
```

## Description

3D property; when used with a model resource whose type is #particle, allows you to get or set the path property of the resource's particle emitter.

This property is a list of vectors that define the path particles follow over their lifetime. The default value of this property is an empty list [].

## Example

In this example, ThermoSystem is a model resource of the type #particle. This statement specifies that the particles of ThermoSystem will follow the path outlined by the list of vectors.

```
member("Fires").modelResource("ThermoSystem").emitter.path = \
    [vector(0,0,0), vector(15,0,0), vector(30,30,-10)]
```

## See also

pathStrength, emitter

# pathName (cast member property)

## Syntax

```
member(whichFlashMember).pathName
the pathName of member whichFlashMember
```

## Description

Cast member property; controls the location of an external file that stores the assets of a Flash movie cast member are stored. You can link a Flash movie to any path on a local or network drive or to a URL.

Setting the path of an unlinked cast member converts it to a linked cast member.

This property can be tested and set. The pathName property of an unlinked member is an empty string.

This property is the same as the fileName property for other member types, and you can use fileName instead of pathName.

## Example

The following startMovie script creates a new Flash cast member using the new command, sets the newly created cast member's linked property so that the cast member's assets are stored in an external file, and then sets the cast member's pathName property to the location of a Flash movie on the World Wide Web:

```
on startMovie
    member(new(#flash)).pathName = \
    "http://www.someURL.com/myFlash.swf"
end
```

## See also

fileName (cast member property), linked

## pathName (movie property)

This is obsolete. Use `moviePath`.

## pathStrength

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
    emitter.pathStrength
```

### Description

3D property; when used with a model resource whose type is `#particle`, determines how closely the particles follow the path specified by the `path` property of the emitter. Its range starts at 0.0 (no strength - so the particles won't be attracted to the path) and continues to infinity. Its default value is 0.1. Setting `pathStrength` to 0.0 is useful for turning the path off temporarily.

As the value of `pathStrength` gets larger, the entire particle system will get more and more stiff. Large `pathStrength` values will tend to make the particles bounce around very quickly, unless some dampening force is also used, such as the `particle drag` property.

This property can be tested and set.

### Example

In this example, `ThermoSystem` is a model resource of the type `#particle`. This statement sets the `pathStrength` property of `ThermoSystem` to 0.97. If a path is outlined by `ThermoSystem`'s `emitter.path` property, the particles follow that path very closely.

```
member("Fires").modelResource("ThermoSystem").emitter.\
    pathStrength = 0.97
```

### See also

`path`, `emitter`

## pattern

### Syntax

```
member(whichCastMember).pattern
the pattern of member whichCastMember
```

### Description

Cast member property; determines the pattern associated with the specified shape. Possible values are the numbers that correspond to the swatches in the Tools window's patterns palette. If the shape cast member is unfilled, the pattern is applied to the cast member's outer edge.

The Director player for Java can assign only the patterns for chips 1 and 15 in the Director Patterns palette.

This property can be useful in Shockwave movies to change images by changing the tiling applied to a shape, allowing you to save memory required by larger bitmaps.

This property can be tested and set.

## Examples

The following statements make the shape cast member Target Area a filled shape and assign it pattern 1, which is a solid color:

```
member("Target Area").filled = TRUE  
member("Target Area").pattern = 1
```

This handler cycles through eight tiles, with each tile's number offset from the previous one, enabling you to create animation using smaller bitmaps:

```
on exitFrame  
  currentPat = member("Background Shape").pattern  
  nextPat = 57 + ((currentPat - 56) mod 8)  
  member("Background Shape").pattern = nextPat  
go the frame  
end
```

## pause (movie playback)

This is obsolete. Use `go to the frame`.

## pause() (3D)

### Syntax

```
member(whichCastmember).model(whichModel).bonesPlayer.pause()  
member(whichCastmember).model(whichModel).keyframePlayer.pause()
```

### Description

3D `#keyframePlayer` and `#bonesPlayer` modifier command; halts the motion currently being executed by the model. Use the `play()` command to unpause the motion.

When a model's motion has been paused by using this command, the model's `bonesPlayer.playing` property will be set to `FALSE`.

### Example

This statement pauses the current animation of the model named Ant3:

```
member("PicnicScene").model("Ant3").bonesplayer.pause()
```

### See also

`play()` (3D), `playing` (3D), `playlist`

## pause (RealMedia)

### Syntax

```
sprite(whichSprite).pause()  
member(whichCastmember).pause()
```

### Description

RealMedia sprite or cast member method; pauses playback of the media stream. The `mediaStatus` value becomes `#paused`.

Calling this method while the RealMedia stream is playing does not change the `currentTime` property and does not clear the media buffer; this allows subsequent `play` commands to resume playback without rebuffering the RealMedia stream.

### Examples

The following examples pause the playing of sprite 2 or the cast member Real.

```
sprite(2).pause()  
member("Real").pause()
```

### See also

mediaStatus, play (RealMedia), seek, stop (RealMedia)

## pause() (sound playback)

### Syntax

```
sound(channelNum).pause()  
pause(sound(channelNum))
```

### Description

This command suspends playback of the current sound in sound channel *channelNum*. A subsequent `play()` command will resume playback.

### Example

This statement pauses playback of the sound cast member playing in sound channel 1:

```
sound(1).pause()
```

### See also

breakLoop(), isBusy(), play() (sound), playNext(), queue(), rewind(), status, stop() (sound)

## pausedAtStart (Flash, digital video)

### Syntax

```
member(whichFlashOrDigitalVideoMember).pausedAtStart  
the pausedAtStart of member whichFlashOrDigitalVideoMember
```

### Description

Cast member property; controls whether the digital video or Flash movie plays when it appears on the Stage. If this property is `TRUE`, the digital video or Flash movie does not play when it appears. If this property is `FALSE`, it plays immediately when it appears.

For a digital video cast member, the property specifies whether the Paused at Start check box in the Digital Video Cast Member Properties dialog box is selected or not.

This property can be tested and set.

### Example

This statement turns on the Paused at Start check box in the Digital Video Cast Member Info dialog box for the QuickTime movie Rotating Chair:

```
member("Rotating Chair").pausedAtStart = TRUE
```

### See also

play

## pausedAtStart (RealMedia)

### Syntax

```
sprite(whichSprite).pausedAtStart  
member(whichCastmember).pausedAtStart
```

### Description

RealMedia sprite or cast member property; allows you to get or set whether a RealMedia stream on the Stage automatically begins to play when buffering is complete (FALSE or 0) or remains paused (TRUE or 1). This property can be set to an expression that evaluates to TRUE or FALSE. Integer values other than 1 or 0 are treated as TRUE. The default setting for this property is FALSE. You can set this property to TRUE by selecting Paused in the graphical view of the Property inspector.

If this property is set to FALSE, the user must click the Play button in the RealMedia viewer (or a button you have created for this purpose in your movie), or you must call the Lingo `play` command to play the sprite when buffering is complete.

This property only affects score-based playback and does not affect playback in the RealMedia viewer.

### Examples

The following examples show that the `pausedAtStart` property of sprite 2 and the cast member Real is set to FALSE, which means that the RealMedia stream will automatically begin to play once buffering is complete.

```
put sprite(2).pausedAtStart  
-- 0  
  
put member("Real").pausedAtStart  
-- 0
```

The following examples set the `pausedAtStart` property for sprite 2 and the cast member Real to TRUE, which means the RealMedia stream will not begin to play unless the `play` command is called.

```
sprite(2).pausedAtStart = TRUE  
member("Real").pausedAtStart = TRUE
```

The following example uses the `pausedAtStart` property to buffer a RealMedia sprite off the Stage, and then play it on the Stage once the buffering is complete. In this example, the RealMedia member has its `pausedAtStart` property set to TRUE. An instance of this member is positioned off the Stage, in sprite channel 1. The following frame script should be placed in the sprite span:

```
on exitFrame me  
  if sprite(1).state > 3 then -- check to see if buffering is complete  
    sprite(1).locH = 162  
    sprite(1).locV = 118  
    sprite(1).play() -- position and play the sprite  
  end if  
end
```

The RealMedia sprite will buffer off the Stage and then appear on the Stage and play immediately when the buffering is complete.

## pause member

### Syntax

```
member(whichCastMember). pause()  
pause member ("whichCastMember")
```

### Description

Command; pauses the streaming of a Shockwave Audio (SWA) streaming cast member. When the sound is paused, the `state` member property equals 4. The portion of the sound that has already been downloaded and is available will continue to play until the cache runs out.

### Example

This handler could be used for a Play or Pause button. If the sound is playing, the handler pauses the sound; otherwise, the handler plays the sound linked to the SWA streaming cast member `soundSWA`.

```
on mouseDown  
  whatState = member("soundSWA").state  
  if whatState = 3 then  
    member("soundSWA").pause()  
  else  
    member("soundSWA").play()  
  end if  
end
```

### See also

`play member`, `stop member`

## pause sprite

### Syntax

```
sprite(whichGIFSpriteNumber). pause()  
pause(sprite whichGIFSpriteNumber)
```

### Description

Command; causes an animated GIF sprite to pause in its playback and remain on the current frame.

### Example

```
sprite(1).pause()
```

### See also

`resume sprite`, `rewind sprite`

## percentBuffered

### Syntax

```
sprite(whichSprite).percentBuffered  
member(whichCastmember).percentBuffered
```

### Description

RealMedia sprite or cast member property; returns the percentage of the buffer that has been filled with the RealMedia stream that is loading from a local file or the server. When this property reaches 100, the buffer is full, and the RealMedia stream begins to play if the `pausedAtStart` property is not set to `TRUE`. This property is dynamic during playback and cannot be set.



The buffer is a type of memory cache that contains the portion of the movie that is about to play, usually just a few seconds. The stream enters the buffer as it streams to RealPlayer and leaves the buffer as RealPlayer plays the clip. The buffer allows viewers to view content without downloading the entire file, and prevents network congestion or lapses in bandwidth availability from disrupting the playback stream.

The buffering process is initiated by the `play` command, and once the buffer is full (100%), the portion of the stream that is in the buffer begins to play. Because the initial buffering process takes a few seconds, there is a delay between the time when the `play` command is called and when the stream actually begins to play. You can work around this using the `pausedAtStart` command, starting to play the stream off the Stage during the buffering process, and then displaying the stream on the Stage as it actually begins to play. (For more information, see the example in the `pausedAtStart (RealMedia)` entry.)

### Examples

The following examples show that 56% of the RealMedia stream in sprite 2 and the cast member Real has been buffered.

```
put sprite(2).percentBuffered
-- 56

put member("Real").percentBuffered
-- 56
```

### See also

`mediaStatus`, `pausedAtStart (RealMedia)`, `state (RealMedia)`

## pauseState

### Syntax

the `pauseState`

### Description

Movie property; determines whether the pause command is currently pausing the movie (TRUE) or not (FALSE).

Because the `pause` command is obsolete, this property is not commonly used.

### Example

This statement checks whether the movie is currently paused and, if it is paused, causes the movie to continue playing:

```
if the pauseState = TRUE then go the frame + 1
```

### See also

`pause (movie playback)`

## percentPlayed

### Syntax

`member(whichCastMember).percentPlayed`  
the `percentPlayed` of member *whichCastMember*

### Description

Shockwave Audio (SWA) cast member property; returns the percentage of the specified SWA file that has actually played.

This property can be tested only after the SWA sound starts playing or has been preloaded by means of the `preLoadBuffer` command. This property cannot be set.

#### Example

This handler displays the percentage of the SWA streaming cast member Frank Sinatra that has played and puts the value in the field cast member Percent Played:

```
on exitFrame
    whatState = member("Frank Sinatra").state
    if whatState > 1 AND whatState < 9 then
        member("Percent Played").text = /
string(member("Frank Sinatra").percentPlayed)
    end if
end
```

#### See also

`percentStreamed`

## percentStreamed

#### Syntax

```
member(whichSWAorFlashCastMember).percentStreamed  
the percentStreamed of member whichSWAorFlashCastMember  
sprite(whichQuickTimeSprite).percentStreamed
```

#### Description

Shockwave Audio (SWA) and Flash cast member property, and QuickTime sprite property.

For SWA streaming sounds, gets the percent of a SWA file already streamed from an HTTP or FTP server. For SWA, this property differs from the `percentPlayed` property in that it includes the amount of the file that has been buffered but not yet played. This property can be tested only after the SWA sound starts playing or has been preloaded by means of the `preLoadBuffer` command.

For Flash movie cast members, this property gets the percent of a Flash movie that has streamed into memory.

For QuickTime sprites, this property gets the percent of the QuickTime file that has played.

This property can have a value from 0 to 100%. For a file on a local disk, the value is 100. For files being streamed from the Internet, the `percentStreamed` value increases as more bytes are received. This property cannot be set.

#### Example

This example displays the percentage of the SWA streaming cast member Ray Charles that has streamed and puts the value in a field:

```
on exitFrame
    whatState = member("Ray Charles").state
    if whatState > 1 AND whatState < 9 then
        member("Percent Streamed Displayer").text = \
string(member("Ray Charles").percentStreamed)
    end if
end
```

This frame script keeps the playhead looping in the current frame so long as less than 60 percent of a Flash movie called Splash Screen has streamed into memory:

```
on exitFrame
  if member("Splash Screen").percentStreamed < 60 then
    go to the frame
  end if
end
```

**See also**

percentPlayed

## percentStreamed (3D)

**Syntax**

member(*whichCastMember*).percentStreamed

**Description**

3D property; allows you to get the percentage of a 3D cast member that has been streamed. This property refers to either the initial file import or to the last file load requested. The value returned is an integer and has a range from 0 to 100. There is no default value for this property.

**Example**

This statement shows that the cast member PartyScene has finished loading.

```
put member("PartyScene").percentStreamed
-- 100
```

## period

**Syntax**

*timeoutObject*.period

**Description**

Object property; the number of milliseconds between timeout events sent by the `timeoutObject` to the timeout handler.

This property can be tested and set.

**Example**

This timeout handler decreases the timeout's `period` by one second each time it's invoked, until a minimum period of 2 seconds (2000 milliseconds) is reached:

```
on handleTimeout timeoutObject
  if timeoutObject.period > 2000 then
    timeoutObject.period = timeoutObject.period - 1000
  end if
end handleTimeout
```

**See also**

name (timeout property), persistent, target, time (timeout object property), timeout(), timeoutHandler, timeoutList

## perpendicularTo

### Syntax

*vector1*.perpendicularTo(*vector2*)

### Description

3D vector command; returns a vector perpendicular to both the original vector and a second vector (*vector2*). This command is equivalent to the vector `crossProduct` command. See `crossProduct()`.

### Example

In this example, *pos1* is a vector on the x axis and *pos2* is a vector on the y axis. The value returned by `pos1.perpendicularTo(pos2)` is `vector( 0.0000, 0.0000, 1.00000e4 )`. The last two lines of the example show the vector which is perpendicular to both *pos1* and *pos2*.

```
pos1 = vector(100, 0, 0)
pos2 = vector(0, 100, 0)
put pos1.perpendicularTo(pos2)
-- vector( 0.0000, 0.0000, 1.00000e4 )
```

### See also

`crossProduct()`, `cross`

## persistent

### Syntax

*timeoutObject*.persistent

### Description

Object property; determines whether the given *timeoutObject* is removed from the *timeoutList* when the current movie stops playing. If `TRUE`, *timeoutObject* remains active. If `FALSE`, the timeout object is deleted when the movie stops playing. The default value is `FALSE`.

Setting this property to `TRUE` allows a timeout object to continue generating timeout events in other movies. This is useful when one movie branches to another with the `go to movie` command.

### Example

This `prepareMovie` handler creates a timeout object that will remain active after the declaring movie stops playing:

```
on prepareMovie
-- Make a timeout object that sends an event every 60 minutes.
  timeout("reminder").new(1000 * 60 * 60, #handleReminder)
  timeout("reminder").persistent = TRUE
end
```

### See also

`name` (timeout property), `period`, `target`, `time` (timeout object property), `timeout()`, `timeoutHandler`, `timeoutList`

# PI

## Syntax

PI

## Description

Constant; returns the value of pi ( $\pi$ ), the ratio of a circle's circumference to its diameter, as a floating-point number. The value is rounded to the number of decimal places set by the `floatPrecision` property.

## Example

This statement uses the PI constant as part of an equation for calculating the area of a circle:

```
set vArea = PI*power(vRadius,2)
```

## picture (cast member property)

### Syntax

`member(whichCastMember).picture`  
the picture of member *whichCastMember*

### Description

Cast member property; determines which image is associated with a bitmap, text, or PICT cast member. To update changes to a cast member's registration point or update changes to an image after relinking it using the `fileName` property, use the following statement:

```
member(whichCastMember).picture = member(whichCastMember).picture
```

where you replace *whichCastMember* with the name or number of the affected cast member.

Because changes to cast members are stored in RAM, this property is best used during authoring. Avoid setting it in projectors.

The property can be tested and set.

### Example

This statement sets the variable named `pictHolder` to the image in the cast member named `Sunset`:

```
pictHolder = member("Sunset").picture
```

### See also

`type` (sprite property)

## picture (window property)

### Syntax

`the stage.picture`  
the picture of the stage  
`window whichWindow.picture`  
the picture of window *whichWindow*

### Description

Window property; this property provides a way to get a picture of the current contents of a window (either the Stage window or a movie in a window).

You can apply the resulting bitmap data to an existing bitmap or use it to create a new one.

This property can be read and but not set.

#### Example

This statement grabs the current content of the Stage and places it into a bitmap cast member:

```
member("Stage image").picture = (the stage).picture
```

#### See also

`media`, `picture` (cast member property)

## pictureP()

#### Syntax

```
pictureP(pictureValue)
```

#### Description

Function; reports whether the state of the `picture` member property for the specified cast member is TRUE (1) or FALSE (0).

Because `pictureP` doesn't directly check whether a picture is associated with a cast member, you must test for a picture by checking the cast member's `picture` member property.

#### Example

The first statement in this example assigns the value of the `picture` member property for the cast member `Shrine`, which is a bitmap, to the variable `pictureValue`. The second statement checks whether `Shrine` is a picture by checking the value assigned to `pictureValue`.

```
set pictureValue to the picture of member "Shrine"  
put pictureP(pictureValue)
```

The result is 1, which is the numerical equivalent of TRUE.

## platform

#### Syntax

```
the platform
```

#### Description

System property; indicates the platform type for which the projector was created.

This property can be tested but not set.

Possible values are the following:

Possible value	Corresponding platform
Macintosh,PowerPC	PowerPC Macintosh
Windows,32	Windows 95 or Windows NT

For forward compatibility and to allow for addition of values, it is better to test the platform by using `contains`.

When the movie plays back as a converted Java applet, this property's value indicates the browser and operating system in which the applet is playing. The property's value has the following syntax when the movie plays back as an applet:

```
Java javaVersion, browser, operatingSystem
```

The following are the possible values for this property's parameters:

- *javaVersion*: 1.0 or 1.1
- *browser*: IE, Netscape, or UnknownBrowser
- *operatingSystem*: Macintosh, Windows, or UnknownOS

For example, if an applet is playing in Microsoft Internet Explorer with Java 1.1 in Windows, platform has the value Java 1.1, IE, Windows.

### Example

This statement checks whether a projector was created for Windows 95 or Windows NT:

```
on exitFrame
  if the platform contains "Windows,32" then
    castLib("Win95 Art").name = "Interface"
  end if
end
```

### See also

runMode

## play

### Syntax

```
sprite(whichFlashSprite).play()
play [frame] whichFrame
play movie whichMovie
play frame whichFrame of movie whichMovie
play sprite whichFlashSprite
```

### Description

Command; branches the playhead to the specified frame of the specified movie or starts a Flash movie sprite playing. For the former, the expression *whichFrame* can be either a string marker label or an integer frame number. The expression *whichMovie* must be a string that specifies a movie file. When the movie is in another folder, *whichMovie* must specify a path.

The `play` command is like the `go to` command, except that when the current sequence finishes playing, `play` automatically returns the playhead to the frame where `play` was called.

If `play` is issued from a frame script, the playhead returns to the next frame; if `play` is issued from a sprite script or handler, the playhead returns to the same frame. A play sequence ends when the playhead reaches the end of the movie or when the `play done` command is issued.

To play a movie from a URL, use `downloadNetThing` or `preloadNetThing()` to download the file to a local disk first, and then use `play` to play the movie on the local disk to minimize download time.

You can use the `play` command to play several movies from a single handler. The handler is suspended while each movie plays but resumes when each movie is finished. Contrast this with a series of `go` commands that, when called from a handler, play the first frame of each movie. The handler is not suspended while the movie plays but immediately continues executing.

When `play` is used to play a Flash movie sprite, the Flash movie plays from its current frame if it is stopped or from its first frame if it is already on the last frame.

Each `play` command needs a matching `play done` command to avoid using up memory if the original calling script isn't returned to. To avoid this memory consumption, you can use a global variable to record where the movie should return to.

### Examples

This statement moves the playhead to the marker named `blink`:

```
play "blink"
```

This statement moves the playhead to the next marker:

```
play marker(1)
```

This statement moves the playhead to a separate movie:

```
play movie "My Drive:More Movies:" & newMovie
```

This frame script checks to see if the Flash movie sprite in channel 5 is playing, and if it is not, it starts the movie:

```
on enterFrame
    if not sprite(5).playing then
        sprite(5).play()
    end if
end
```

### See also

`downloadNetThing`

## play() (3D)

### Syntax

```
member(whichCastmember).model(whichModel).bonesPlayer.play()
member(whichCastmember).model(whichModel).keyframePlayer.play()
member(whichCastmember).model(whichModel).bonesPlayer.\
    play(motionName [, looped, startTime, endTime, scale, offset])
member(whichCastmember).model(whichModel).keyframePlayer.\
    play(motionName [, looped, startTime, endTime, scale, offset])
```

### Description

**3D #keyframePlayer and #bonesPlayer command;** initiates or unpauses the execution of a motion.

When a model's motion has been initiated or resumed by using this command, the model's `bonesPlayer.playing` property will be set to `TRUE`.

Use `play()` with no parameters to resume the execution of a motion that has been paused with the `pause()` command.

When `play()` is called and only the *motionName* parameter is specified, the motion will be executed by the model once from beginning to end at the speed set by the modifier's `playRate` property.

The optional parameters of the `play` command are as follows:

*looped* specifies whether the motion plays once (`FALSE`) or continuously (`TRUE`).

*startTime* is measured in milliseconds from the beginning of the motion. When *looped* is `TRUE`, the first iteration of the loop begins at *offset* and ends at *endTime* with all subsequent repetitions of the motion beginning at *startTime* and end at *endTime*.



*endTime* is measured in milliseconds from the beginning of the motion. When *looped* is FALSE, the motion begins at *offset* and ends at *endTime*. When *looped* is TRUE, the first iteration of the loop begins at *offset* and ends at *endTime* with all subsequent repetitions beginning at *startTime* and end at *endTime*. Set *endTime* to -1 if you want the motion to play to the end.

*playRate* is multiplied by the model's *#keyframePlayer* or *#bonesPlayer* modifier's *playRate* property to determine the actual speed of the motion's playback.

*offset* is measured in milliseconds from the beginning of the motion. When *looped* is FALSE, the motion begins at *offset* and ends at *endTime*. When *looped* is TRUE, the first iteration of the loop begins at *offset* and ends at *endTime* with all subsequent repetitions beginning at *startTime* and end at *cropEnd*. You can alternately specify the *offset* parameter with a value of *#synchronized* in order to start the motion at the same relative position in its duration as the currently playing animation is through its own duration.

Using the *play()* command to initiate a motion inserts the motion at the beginning of the modifier's playlist. If this interrupts playback of another motion, the interrupted motion remains in the playlist in the next position after the newly initiated motion. When the newly initiated motion ends (if it is non-looping) or if the *playNext()* command is issued, the interrupted motion will resume playback at the point where it was interrupted.

### Example

This command causes the model named Walker to begin playback of the motion named Fall. After playing this motion, the model will resume playback of any previously playing motion.

```
sprite(1).member.model("Walker").bonesPlayer.play("Fall", 0, \
    0, -1, 1, 0)
```

This command causes the model named Walker to begin playback of the motion named Kick. If Walker is currently executing a motion, it is interrupted by Kick and a section of Kick will play in a continuous loop. The first iteration of the loop will begin 2000 milliseconds from the motion's beginning. All subsequent iterations of the loop will begin 1000 milliseconds from Kick's beginning and will end 5000 milliseconds from Kick's beginning. The rate of playback will be three times the *playRate* property of the model's *bonesPlayer* modifier.

```
sprite(1).member.model("Walker").bonesPlayer.play("Kick", 1, \
    1000, 5000, 3, 2000)
```

### See also

*queue()* (3D), *playNext()* (3D), *playRate*, *playlist*, *pause()* (3D), *removeLast()*, *playing* (3D)

## play (RealMedia)

### Syntax

```
sprite(whichSprite).play()  
member(whichCastmember).play()
```

### Description

RealMedia sprite or cast member method; starts the streaming process for the RealMedia stream if the stream is closed, or resumes playback if the stream is paused. If necessary, the stream will buffer before beginning to play. The *mediaStatus* value becomes *#playing*.

## Examples

The following examples start the streaming process for the stream in sprite 2 and the cast member Real.

```
sprite(2).play()  
member("Real").play()
```

## See also

`mediaStatus`, `pause (RealMedia)`, `seek`, `stop (RealMedia)`

# play() (sound)

## Syntax

```
sound(channelNum).play()  
sound(channelNum).play(member (whichMember))  
sound(channelNum).play([#member: member(whichmember), {#startTime:  
    milliseconds, #endTime: milliseconds, #loopCount: numberOfLoops,  
    #loopStartTime: milliseconds, #loopEndTime: milliseconds, #preloadTime:  
    milliseconds}])
```

## Description

This function begins playing any sounds queued in *soundObject*, or queues and begins playing the given member.

Sound members take some time to load into RAM before they can begin playback. It's recommended that you queue sounds with `queue()` before you want to begin playing them and then use the first form of this function. The second two forms do not take advantage of the pre-loading accomplished with the `queue()` command.

By using an optional property list, you can specify exact playback settings for a sound. These properties may be optionally set:

Property	Description
#member	The sound cast member to queue. This property must be provided; all others are optional.
#startTime	The time within the sound at which playback begins, in milliseconds. The default is the beginning of the sound. See <code>startTime</code> .
#endTime	The time within the sound at which playback ends, in milliseconds. The default is the end of the sound. See <code>endTime</code> .
#loopCount	The number of times to play a loop defined with <code>#loopStartTime</code> and <code>#loopEndTime</code> . The default is 1. See <code>loopCount</code> .
#loopStartTime	The time within the sound to begin a loop, in milliseconds. See <code>loopStartTime</code> .
#loopEndTime	The time within the sound to end a loop, in milliseconds. See <code>loopEndTime</code> .
#preloadTime	The amount of the sound to buffer before playback, in milliseconds. See <code>preloadTime</code> .

To see an example of `play() (sound)` used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

## Examples

This statement plays cast member `introMusic` in sound channel 1:

```
sound(1).play(member("introMusic"))
```

The following statement plays cast member `creditsMusic` in sound channel 2. Playback begins 4 seconds into the sound and ends 15 seconds into the sound. The section from 10.5 seconds to 14 seconds loops 6 times.

```
sound(2).play([#member: member("creditsMusic"), #startTime: 4000, \
#endTime: 15000, #loopCount: 6, #loopStartTime: 10500, #loopEndTime: 14000])
```

**See also**

`setPlaylist()`, `isBusy()`, `pause()` (sound playback), `playNext()`, `preLoadTime`, `queue()`, `rewind()`, `stop()` (sound)

## playBackMode

**Syntax**

```
sprite(whichFlashSprite).playBackMode
the playBackMode of sprite whichFlashSprite
member(whichGIFAnimMember).playBackMode
the playBackMode of member whichGIFAnimMember
```

**Description**

Cast member and sprite property; controls the tempo of a Flash movie or animated GIF cast member with the following values:

- `#normal` (*default*)—Plays the Flash movie or GIF file as close to the original tempo as possible.
- `#lockStep`—Plays the Flash movie or GIF file frame for frame with the Director movie.
- `#fixed`—Plays the Flash movie or GIF file at the rate specified by the `fixedRate` property.

This property can be tested and set.

**Example**

This sprite script sets the frame rate of a Flash movie sprite to match the frame rate of the Director movie:

```
property spriteNum
on beginSprite me
    sprite(spriteNum).playBackMode = #lockStep
end
```

**See also**

`fixedRate`

## play done

**Syntax**

```
play done
```

**Description**

Command; ends the sequence started with the most recent `play` command. The `play done` command returns the playhead to where the calling sequence initiated the `play` command. If `play` is issued from a frame script, the playhead returns to the next frame; if `play` is issued from a sprite script, the playhead returns to the same frame.

Each `play` command needs a matching `play done` command to avoid using up memory if the original calling script isn't returned to. To avoid this memory consumption, you can use a global variable to record where the movie should return to.

The `play done` command has no effect in a movie that is playing in a window.

#### Example

This handler returns the playhead to the frame of the movie that was playing before the current movie started:

```
on exitFrame
    play done
end
```

#### See also

`play`

## playing

#### Syntax

`sprite(whichFlashSprite).playing`  
the playing of sprite *whichFlashSprite*

#### Description

Flash sprite property; indicates whether a Flash movie is playing (TRUE) or stopped (FALSE).

This property can be tested but not set.

#### Example

This frame script checks to see if the Flash movie sprite in channel 5 is playing and, if it is not, starts the movie:

```
on enterFrame
    if not sprite(5).playing then
        sprite(5).play()
    end if
end
```

## playing (3D)

#### Syntax

`member(whichCastmember).model(whichModel).keyframePlayer.playing`  
`member(whichCastmember).model(whichModel).bonesPlayer.playing`

#### Description

3D `#keyframePlayer` and `#bonesPlayer` modifier property; indicates whether the modifier's animation playback engine is running (TRUE) or if it's paused (FALSE).

This property can be tested but not set.

#### Example

This statement shows that the `#keyframePlayer` animation playback engine for the model named `Alien3` is currently running.

```
put member("newaliens").model("Alien3").keyframePlayer.playing
-- 1
```

#### See also

`play()` (3D), `pause()` (3D), `playlist`, `queue()` (3D)

# playlist

## Syntax

```
member(whichCastmember).model(whichModel).keyframePlayer.playlist  
member(whichCastmember).model(whichModel).bonesPlayer.playlist
```

## Description

3D `#keyframePlayer` and `#bonesPlayer` modifier property; returns a linear list of property lists, each representing a motion queued for playback by the modifier.

Each property list will have the following properties:

- `#name` is the name of the motion to be played.
- `#loop` indicates whether the motion's playback should be looped.
- `#startTime` is the time, in milliseconds at which playback of the animation should begin.
- `#endTime` is the time, in milliseconds at which playback of the animation ends or when the motion should be looped. A negative value indicates that the motion should be played to the end.
- `#scale` is rate of play for the motion that is to be multiplied by the modifier's `playRate` property to determine the actual speed of the motion's playback.

The `playlist` property can be tested but not set. Use the `queue()`, `play()`, `playNext()`, and `removeLast()` commands to manipulate it.

## Example

The following statement displays the currently queued motions for the model `Stroller` in the Message window. There are two currently queued motions: `Walk` and `Jump`.

```
put member("ParkScene").model("Stroller").bonesPlayer.playlist  
-- [[#name: "Walk", #loop: 1, #startTime: 1500, #endTime: 16000, \  
    #scale:1.0000, #offset: 0], [#name: "Jump", #loop: 1, \  
    #startTime: 0, #endTime: 1200, #scale: 1.0000, #offset: 0]]
```

## See also

`play()` (3D), `playNext()` (3D), `removeLast()`, `queue()` (3D)

# play member

## Syntax

```
member(whichCastMember).play()  
play member whichCastMember
```

## Description

Command; begins playback of a Shockwave Audio (SWA) streaming cast member.

If the sound has not been preloaded by means of the `preLoadBuffer` command, the SWA sound preloads before playing begins. When the sound is playing, the `state` member property equals 3.

Be aware that Xtra extensions to support this functionality must be included when playing back a streaming sound.

**Example**

This handler begins the playback of the cast member Big Band:

```
on mouseDown
    member("Big Band").play()
end
```

**See also**

pause member, stop member

## playNext()

**Syntax**

```
sound(channelNum).playNext()
playNext(sound(channelNum))
```

**Description**

This command immediately interrupts playback of the current sound playing in the given sound channel and begins playing the next queued sound. If no more sounds are queued in the given channel, the sound simply stops playing.

**Example**

This statement plays the next queued sound in sound channel 2:

```
sound(2).playNext()
```

**See also**

pause() (sound playback), play() (sound), stop() (sound)

## playNext() (3D)

**Syntax**

```
member(whichMember).model(whichModel).bonesPlayer.playNext()
member(whichMember).model(whichModel).keyframePlayer.playNext()
```

**Description**

3D `#keyframePlayer` and `#bonesPlayer` modifier command; initiates playback of the next motion in the playlist of the model's `#keyframePlayer` or `#bonesPlayer` modifier. The currently playing motion, which is the first entry in the playlist, is interrupted and removed from the playlist.

If motion blending is enabled, and there are two or more motions in the playlist, blending between the current motion and the next one in the playlist will begin when `playNext()` is called.

**Example**

This statement interrupts the motion currently being executed by model 1 and initiates playback of the next motion in the playlist:

```
member("scene").model[1].bonesPlayer.playnext()
```

**See also**

blend (3D), playlist

# playRate

## Syntax

```
member(whichCastmember).model(whichModel).bonesPlayer.playRate  
member(whichCastmember).model(whichModel).keyframePlayer.playRate
```

## Description

3D `#keyframePlayer` and `#bonesPlayer` modifier property; scale multiplier for the local time of motions being played. This property only partially determines the speed at which motions are executed by the model.

The playback of a motion by a model is the result of either a `play()` or `queue()` command. The `scale` parameter of the `play()` or `queue()` command is multiplied by the modifier's `playRate` property, and the resulting value is the speed at which the particular motion will be played back.

## Example

This statement sets the `playRate` property of the `keyframePlayer` modifier for the model named `GreenAlien` to 3:

```
member("newAliens").model("GreenAlien").keyframePlayer.playRate = 3
```

## See also

`play()` (3D), `queue()` (3D), `playlist`, `currentTime` (3D)

# point()

## Syntax

```
point(horizontal, vertical)
```

## Description

Function and data type; yields a point that has the horizontal coordinate specified by *horizontal* and the vertical coordinate specified by *vertical*.

A point has a `locH` and a `locV` property. Point coordinates can be changed by arithmetic operations.

To see an example of `point()` used in a completed movie, see the Imaging and Vector Shapes movies in the Learning/Lingo Examples folder inside the Director application folder.

## Examples

This statement sets the variable `lastLocation` to the point (250, 400):

```
set lastLocation = point(250, 400)
```

This statement adds 5 pixels to the horizontal coordinate of the point assigned to the variable `myPoint`:

```
myPoint.locH = myPoint.locH + 5
```

The following statements set a sprite's Stage coordinates to `mouseH` and `mouseV` plus 10 pixels. The two statements are equivalent.

```
sprite(the clickOn).loc = point(the mouseH, the mouseV) + point(10, 10)  
sprite(the clickOn).loc = the mouseLoc + 10
```

This handler moves a named sprite to the location that the user clicks:

```
end mouseDown

on mouseDown
-- Set these variables as needed for your own movie
theSprite = 1 -- Set the sprite that should move
steps = 40 -- Set the number of steps to get there
initialLoc = sprite(theSprite).loc
delta = (the clickLoc - initialLoc) / steps
repeat with i = 1 to steps
    sprite(theSprite).loc = initialLoc + (i * delta)
    updateStage
end repeat
end mouseDown
```

**See also**

`mouseLoc`, `flashToStage()`, `rect()`, `stageToFlash()`

## pointAt

**Syntax**

```
member(whichCastmember).model(whichModel).pointAt\
    (vectorPosition{, vectorUp})
member(whichCastmember).camera(whichCamera).pointAt\
    (vectorPosition{, vectorUp})
member(whichCastmember).light(whichLight).pointAt\
    (vectorPosition{, vectorUp})
member(whichCastmember).group(whichGroup).pointAt\
    (vectorPosition{, vectorUp})
```

**Description**

3D command; rotates the referenced object so that its forward direction vector points at the world relative position specified by *vectorPosition*, then it rotates the referenced object to point its up direction vector in the direction hinted at by the world relative vector specified by *vectorUp*. The value of *vectorPosition* can also be a node reference.

The optional parameter *vectorUp* is a world relative vector that hints at where the object's up vector should point. If this parameter isn't specified, then this command defaults to using the world's y axis as the up hinting vector. If you attempt to point the object at a position such that the object's forward vector is parallel to the world's y axis, then the world's x axis is used as the up hinting vector.

The direction at which you wish to point the object's forward direction and the direction specified by *vectorUp* do not need to be perpendicular to each other being as this command only uses the *vectorUp* parameter as a hinting vector.

The object's front and up direction vectors are defined by the object's `pointAtOrientation` property.



## Examples

This example points three objects at the model named Mars: the camera named MarsCam, the light named BrightSpot, and the model named BigGun:

```
thisWorldPosn = member("Scene").model("Mars").worldPosition
member("Scene").camera("MarsCam").pointAt(thisWorldPosn)
member("Scene").light("BrightSpot").pointAt(thisWorldPosn)
member("Scene").model("BigGun").pointAt(thisWorldPosn, \
    vector(0,0,45))
```

If you use non-uniform scaling and a custom `pointAtOrientation` on the same node, e.g., a model, using `pointAt` will likely cause unexpected non-uniform scaling. This is due to the order in which the non-uniform scaling and the rotation to properly orient the node are applied. To workaround this issue, do one of the following:

- Avoid using non-uniform scaling and non-default `pointAtOrientation` together on the same node.
- Remove your scale prior to using `pointAt`, and then reapply it afterwards.

For example:

```
scale = node.transform.scale
node.scale = vector(1, 1, 1)
node.pointAt(vector(0, 0, 0)) -- non-default pointAtOrientation
node.transform.scale = scale
```

## See also

`pointAtOrientation`

# pointAtOrientation

## Syntax

```
member(whichCastmember).model(whichModel).pointAtOrientation
member(whichCastmember).group(whichGroup).pointAtOrientation
member(whichCastmember).light(whichLight).pointAtOrientation
member(whichCastmember).camera(whichCamera).pointAtOrientation
```

## Description

3D model, light, group and camera property; allows you to get or set how the referenced object responds to the `pointAt` command. This property is a linear list of two object-relative vectors, the first vector in the list defines which direction is considered the object's front direction, the second defines which direction is considered the object's up direction.

The object's front and up directions do not need to be perpendicular to each other, but they should not be parallel to each other.

## Example

This statement displays the object-relative front direction and up direction vectors of the model named `bip01`:

```
put member("scene").model("bip01").pointAtOrientation
-- [vector(0.0000, 0.0000, 1.0000), vector(0.0000, 1.0000, 0.0000)]
```

## See also

`pointAt`

## pointInHyperlink()

### Syntax

```
sprite(whichSpriteNumber).pointInHyperlink(point)  
pointInHyperlink(sprite whichSpriteNumber, point)
```

### Description

Text sprite function; returns a value (TRUE or FALSE) that indicates whether the specified point is within a hyperlink in the text sprite. Typically, the point used is the cursor position. This is useful for setting custom cursors.

### See also

`cursor (command)`, `mouseLoc`

## pointOfContact

### Syntax

```
collisionData.pointOfContact
```

### Description

3D `collisionData` property; returns a vector describing the point of contact in a collision between two models.

The `collisionData` object is sent as an argument with the `#collideWith` and `#collideAny` events to the handler specified in the `registerForEvent`, `registerScript`, and `setCollisionCallback` commands.

The `#collideWith` and `#collideAny` events are sent when a collision occurs between models to which collision modifiers have been added. The `resolve` property of the models' modifiers must be set to `TRUE`.

This property can be tested but not set.

### Example

This example has two parts. The first part is the first line of code, which registers the `#explode` handler for the `#collideAny` event. The second part is the `#explode` handler. When two models in the cast member `MyScene` collide, the `#explode` handler is called and the `collisionData` argument is sent to it. The first nine lines of the `#explode` handler create the model resource named `SparkSource` and set its properties. This model resource is a single burst of particles. The tenth line of the handler creates a model named `SparksModel` using the model resource named `SparkSource`. The last line of the handler sets the position of `SparksModel` to the position where the collision occurred. The overall effect is a burst of sparks caused by a collision.

```
member("MyScene").registerForEvent(#collideAny, #explode, 0)

on explode me, collisionData
  nmr = member("MyScene").newModelResource("SparkSource", #particle)
  nmr.emitter.mode = #burst
  nmr.emitter.loop = 0
  nmr.emitter.minSpeed = 30
  nmr.emitter.maxSpeed = 50
  nmr.emitter.direction = vector(0, 0, 1)
  nmr.colorRange.start = rgb(0, 0, 255)
  nmr.colorRange.end = rgb(255, 0, 0)
  nmr.lifetime = 5000
  nm = member("MyScene").newModel("SparksModel", nmr)
  nm.transform.position = collisionData.pointOfContact
end
```

### See also

`modelA`, `modelB`

## pointToChar()

### Syntax

```
pointToChar(sprite spriteNumber, pointToTranslate)
```

### Description

Function; returns an integer representing the character position located within the text or field `sprite spriteNumber` at screen coordinate `pointToTranslate`, or returns -1 if the point is not within the text.

This function can be used to determine the character under the cursor.

### Example

This Lingo displays the number of the character being clicked, as well as the letter, in the Message window:

```
property spriteNum

on mouseDown me
  pointClicked = the mouseLoc
  currentMember = sprite(spriteNum).member
  charNum = sprite(spriteNum).pointToChar(pointClicked)
  actualChar = currentMember.char[charNum]
  put "Clicked character" && charNum & ", the letter" && actualChar
end
```

### See also

`mouseLoc`, `pointToWord()`, `pointToItem()`, `pointToLine()`, `pointToParagraph()`

## pointToItem()

### Syntax

```
sprite(whichSpriteNumber).pointToItem(pointToTranslate)  
pointToItem(sprite spriteNumber, pointToTranslate)
```

### Description

Function; returns an integer representing the item position in the text or field sprite *spriteNumber* at screen coordinate *pointToTranslate*, or returns -1 if the point is not within the text. Items are separated by the `itemDelimiter` property, which is set to a comma by default.

This function can be used to determine the item under the cursor.

### Example

This Lingo displays the number of the item being clicked, as well as the text of the item, in the Message window:

```
property spriteNum  
  
on mouseDown me  
    pointClicked = the mouseLoc  
    currentMember = sprite(spriteNum).member  
    itemNum = sprite(spriteNum).pointToItem(pointClicked)  
    itemText = currentMember.item[itemNum]  
    put "Clicked item" && itemNum & ", the text" && itemText  
end
```

### See also

`itemDelimiter`, `mouseLoc`, `pointToChar()`, `pointToWord()`, `pointToItem()`,  
`pointToLine()`, `pointToParagraph()`

## pointToLine()

### Syntax

```
sprite(whichSpriteNumber).pointToLine(pointToTranslate)  
pointToLine(sprite spriteNumber, pointToTranslate)
```

### Description

Function; returns an integer representing the line position in the text or field sprite *spriteNumber* at screen coordinate *pointToTranslate*, or returns -1 if the point is not within the text. Lines are separated by carriage returns in the text or field cast member.

This function can be used to determine the line under the cursor.

### Example

This Lingo displays the number of the line being clicked, as well as the text of the line, in the Message window:

```
property spriteNum
```

```
on mouseDown me  
    pointClicked = the mouseLoc  
    currentMember = sprite(spriteNum).member  
    lineNum = sprite(spriteNum).pointToLine(pointClicked)  
    lineText = currentMember.line[lineNum]  
    put "Clicked line" && lineNum & ", the text" && lineText  
end
```

### See also

```
itemDelimiter, mouseLoc, pointToChar(), pointToWord(), pointToItem(),  
pointToLine(), pointToParagraph()
```

## pointToParagraph()

### Syntax

```
sprite(whichSpriteNumber).pointToParagraph(pointToTranslate)  
pointToParagraph(sprite spriteNumber, pointToTranslate)
```

### Description

Function; returns an integer representing the paragraph number located within the text or field sprite *spriteNumber* at screen coordinate *pointToTranslate*, or returns -1 if the point is not within the text. Paragraphs are separated by carriage returns in a block of text.

This function can be used to determine the paragraph under the cursor.

### Example

This Lingo displays the number of the paragraph being clicked, as well as the text of the paragraph, in the message window:

```
property spriteNum
```

```
on mouseDown me  
    pointClicked = the mouseLoc  
    currentMember = sprite(spriteNum).member  
    paragraphNum = sprite(spriteNum).pointToParagraph(pointClicked)  
    paragraphText = currentMember.paragraph[paragraphNum]  
    put "Clicked paragraph" && paragraphNum & ", the text" && paragraphText  
end
```

#### See also

`itemDelimiter`, `mouseLoc`, `pointToChar()`, `pointToWord()`, `pointToItem()`, `pointToLine()`

## pointToWord()

#### Syntax

```
sprite(whichSpriteNumber).pointToWord(pointToTranslate)  
pointToWord(sprite spriteNumber, pointToTranslate)
```

#### Description

Function; returns an integer representing the number of a word located within the text or field `sprite spriteNumber` at screen coordinate *pointToTranslate*, or returns -1 if the point is not within the text. Words are separated by spaces in a block of text.

This function can be used to determine the word under the cursor.

#### Example

This Lingo displays the number of the word being clicked, as well as the text of the word, in the Message window:

```
property spriteNum  
  
on mouseDown me  
    pointClicked = the mouseLoc  
    currentMember = sprite(spriteNum).member  
    wordNum = sprite(spriteNum).pointToWord(pointClicked)  
    wordText = currentMember.word[wordNum]  
    put "Clicked word" && wordNum & ", the text" && wordText  
end
```

#### See also

`itemDelimiter`, `mouseLoc`, `pointToChar()`, `pointToItem()`, `pointToLine()`, `pointToParagraph()`

## position (transform)

#### Syntax

```
member(whichCastmember).node(whichNode).transform.position  
member(whichCastmember).node(whichNode).getWorldTransform().\  
    position  
transform.position
```

#### Description

3D property; allows you to get or set the positional component of a transform. A transform defines a scale, position and rotation within a given frame of reference. The default value of this property is vector(0,0,0).

A node can be a camera, group, light or model object. Setting the `position` of a node's transform defines that object's position within the transform's frame of reference. Setting the `position` property of an object's world relative transform using `getWorldTransform().position` defines the object's position relative to the world origin. Setting the `position` property of an object's parent relative transform using `transform.position` defines the object's position relative to its parent node.

The `worldPosition` property of a model, light, camera or group object is a shortcut to the `getWorldTransform().position` version of this property for that object.

### Examples

The following statement displays the parent-relative position of the model named Tire.

```
put member("scene").model("Tire").transform.position
-- vector(-15.000, -2.5000, 20.0000)
```

The following statement displays the world-relative position of the model named Tire.

```
put member("scene").model("Tire").getWorldTransform().position
-- vector(5.0000, -2.5000, -10.0000)
```

The following statements first store the world transform of the model named Tire in the variable tempTransform, then they display the position component of that transform.

```
tempTransform = member("scene").model("Tire").getWorldTransform()
put tempTransform.position
-- vector(5.0000, -2.5000, -10.0000)
```

### See also

transform (property), getWorldTransform(), rotation (transform), scale (transform)

## positionReset

### Syntax

```
member(whichCastmember).model(whichModel).bonesPlayer.\
    positionReset
member(whichCastmember).model(whichModel).keyframePlayer.\
    positionReset
```

### Description

3D keyframePlayer and bonesPlayer modifier property; indicates whether the model returns to its starting position after the end of a motion (TRUE) or not (FALSE).

The default value for this property is TRUE.

### Example

This statement prevents the model Monster from returning to its original position when it finishes the execution of a motion:

```
member("NewAlien").model("Monster").keyframePlayer.\
    positionReset = FALSE
```

### See also

currentLoopState

## posterFrame

### Syntax

```
member(whichFlashMember).posterFrame
the posterFrame of member whichFlashMember
```

### Description

Flash cast member property; controls which frame of a Flash movie cast member is used for its thumbnail image. This property specifies an integer corresponding to a frame number in the Flash movie.

This property can be tested and set. The default value is 1.

### Example

This handler accepts a reference to a Flash movie cast member and a frame number as parameters, and it then sets the thumbnail of the specified movie to the specified frame number:

```
on resetThumbnail whichFlashMovie, whichFrame  
    member(whichFlashMovie).posterFrame = whichFrame  
end
```

## postNetText

### Syntax

```
postNetText(url, propertyList {,serverOSSString} {,serverCharSetString})  
postNetText(url, postText {,serverOSSString} {,serverCharSetString})
```

### Description

Command; sends a POST request to *url*, which is an HTTP URL, with *postText* as the data.

This command is similar to `getNetText()`. As with `getNetText()`, the server's response is returned by `netTextResult(netID)` once `netDone(netID)` becomes 1, and if `netError(netID)` is 0, or okay.

When a property list is used instead of a string, the information is sent in the same way a browser posts an HTML form, with `METHOD=POST`. This facilitates the construction and posting of form data within a Director title. Property names correspond to HTML form field names and property values to field values.

The property list can use either strings or symbols as the property names. If a symbol is used, it is automatically converted to a string without the # at the beginning. Similarly, a numeric value is converted to a string when used as the value of a property.

**Note:** If a program uses the alternate form—a string instead of property list—the string *postText* is sent to the server as an HTTP POST request using MIME type "text/plain." This will be convenient for some applications, but is not compatible with HTML forms posting.

The optional parameter *serverOSSString* defaults to UNIX but may be set to Windows or Mac and translates any carriage returns in the *postText* argument into those used on the server to avoid confusion. For most applications, this setting is unnecessary because line breaks are usually not used in form responses.

The optional parameter *serverCharSetString* applies only if the user is running on a Shift-JIS (Japanese) system. Its possible settings are "JIS", "EUC", "ASCII", and "AUTO". Posted data is converted from Shift-JIS to the named character set. Returned data is handled exactly as by `getNetText()` (converted from the named character set to Shift-JIS). If you use "AUTO", the posted data from the local character set is not translated; the results sent back by the server are translated as they are for `getNetText()`. "ASCII" is the default if *serverCharSetString* is omitted. "ASCII" provides no translation for posting or results.

The optional arguments may be omitted without regard to position.

This command also has an additional advantage over `getNetText()`: a `postNetText()` query can be arbitrarily long, whereas the `getNetText()` query is limited to the length of a URL (1K or 4K, depending on the browser).

**Note:** If you use `postNetText` to post data to a domain different from the one the movie is playing from, the movie will display a security alert when playing back in Shockwave.

To see an example of `postNetText` used in a completed movie, see the Forms and Post movie in the Learning/Lingo Examples folder inside the Director application folder.



### Examples

This statement omits the *serverCharSetString* parameter:

```
netID = postNetText("www.mydomain.com\database.cgi", "Bill Jones", "Win")
```

This example generates a form from user-entry fields for first and last name, along with a Score.

Both *serverOSSString* and *serverCharSetString* have been omitted:

```
lastName = member("Last Name").text
firstName = member("First Name").text
totalScore = member("Current Score").text
infoList = ["FName":firstName, "LName":lastName, "Score":totalScore]
netID = postNetText("www.mydomain.com\userbase.cgi", infoList)
```

### See also

`getNetText()`, `netTextResult()`, `netDone()`, `netError()`

## power()

### Syntax

```
power(base, exponent)
```

### Description

Math function; calculates the value of the number specified by *base* to the exponent specified by *exponent*.

### Example

This statement sets the variable `vResult` to the value of 4 to the third power:

```
set vResult = power(4,3)
```

## preferred3DRenderer

### Syntax

```
the preferred3DRenderer
```

### Description

3D movie property; allows you to get or set the default renderer used to draw 3D sprites within a particular movie if that renderer is available on the client machine. If the specified renderer is not available on the client machine, the movie selects the most suitable available renderer.

The possible values for this property are as follows:

`#openGL` specifies the openGL drivers for a hardware acceleration that work with both Macintosh and Windows platforms.

`#directX7_0` specifies the DirectX 7 drivers for hardware acceleration that work only with Windows platforms.

`#directX5_2` specifies the DirectX 5.2 drivers for hardware acceleration that work only with Windows platforms.

`#software` specifies the Director built-in software renderer that works with both Macintosh and Windows platforms.

`#auto` specifies that the most suitable renderer should be chosen. This is the default value for this property.

The value set for this property is used as the default for the `Renderer Services` object's `renderer` property.

This property differs from the `getRendererServices()` object's `renderer` property in that the `preferred3dRenderer` specifies the preferred renderer to use, whereas the `getRendererServices()` object's `renderer` property indicates what renderer is actually being used by the movie.

Shockwave users have the option of specifying the renderer of their choice using the 3D Renderer context menu in Shockwave. If the user selects the “Obey content settings” option, the renderer specified by the `renderer` or `preferred3dRenderer` property is used to draw the movie (if available on the user's system), otherwise, the renderer selected by the user is used.

#### Example

This statement allows the movie to pick the best 3D renderer available on the user's system:

```
the preferred3dRenderer = #auto
```

#### See also

`renderer`, `getRendererServices()`, `rendererDeviceList`

## preLoad (3D)

#### Syntax

```
member(whichCastmember).preload  
memberReference.preload
```

#### Description

3D property; allows you to get or set whether data is preloaded before playing (`TRUE`), or is streamed while playing (`FALSE`). This property can be used only with linked files. The default value is `FALSE`.

In Director, setting the `preLoad` property to `TRUE` causes the cast member to load completely before playback starts. In Shockwave, setting the `preLoad` property to `TRUE` causes the cast member to begin streaming when the movie starts playing. Before performing any Lingo operations on a 3D cast member that is streaming, be sure to check that the cast member's `state` property has a value greater than or equal to 2.

#### Example

This statement sets the `preload` property of the cast member `PartyScene` to `FALSE`, which allows externally linked media to stream into `PartyScene` during playback:

```
ember("PartyScene").preload = FALSE  
member("3D world").preload
```

#### See also

`state (3D)`

## preLoad (command)

### Syntax

```
preLoad  
preLoad toFrameNum  
preLoad fromFrame, toFrameNum
```

### Description

Command; preloads cast members in the specified frame or range of frames into memory and stops when memory is full or when all of the specified cast members have been preloaded, as follows:

- When used without arguments, the command preloads all cast members used from the current frame to the last frame of a movie.
- When used with one argument, *toFrame*, the command preloads all cast members used in the range of frames from the current frame to the frame *toFrameNum*, as specified by the frame number or label name.
- When used with two arguments, *fromFrame* and *toFrameNum*, preloads all cast members used in the range of frames from the frame *fromFrame* to the frame *toFrameNum*, as specified by the frame number or label name.

The `preLoad` command also returns the number of the last frame successfully loaded. To obtain this value, use the `result` function.

### Examples

This statement preloads the cast members used from the current frame to the frame that has the next marker:

```
preLoad marker (1)
```

This statement preloads the cast members used from frame 10 to frame 50:

```
preLoad 10, 50
```

### See also

`preLoadMember`

## preLoad (cast member property)

### Syntax

```
member(whichCastMember).preLoad  
the preLoad of member whichCastMember
```

### Description

Cast member property; determines whether the digital video cast member specified by *whichCastMember* can be preloaded into memory (TRUE) or not (FALSE, default). The TRUE status has the same effect as selecting Enable Preload in the Digital Video Cast Member Properties dialog box.

For Flash movie cast members, this property controls whether a Flash movie must load entirely into RAM before the first frame of a sprite is displayed (TRUE), or whether the movie can stream into memory as it plays (FALSE, default). This property works only for linked Flash movies whose assets are stored in an external file; it has no effect on members whose assets are stored in the cast. The `streamMode` and `bufferSize` properties determine how the cast member is streamed into memory.

This property can be tested and set.

### Examples

This statement reports in the Message window whether the QuickTime movie Rotating Chair can be preloaded into memory:

```
put member("Rotating Chair").preload
```

This `startMovie` handler sets up a Flash movie cast member for streaming and then sets its `bufferSize` property:

```
on startMovie
    member("Flash Demo").preload = FALSE
    member("Flash Demo").bufferSize = 65536
end
```

### See also

`bufferSize`, `streamMode`

## preloadBuffer member

### Syntax

```
member(whichCastMember).preloadBuffer()
preloadBuffer member whichCastMember
```

### Description

Command; preloads part of a specified Shockwave Audio (SWA) file into memory. The amount preloaded is determined by the `preloadTime` property. This command works only if the SWA cast member is stopped.

When the `preloadBuffer` command succeeds, the `state` member property equals 2.

Most SWA cast member properties can be tested only after the `preloadBuffer` command has completed successfully. These properties include: `cuePointNames`, `cuePointTimes`, `currentTime`, `duration`, `percentPlayed`, `percentStreamed`, `bitRate`, `sampleRate`, and `numChannels`.

### Example

This statement loads the cast member Mel Torme into memory:

```
member("Mel Torme").preloadBuffer()
```

### See also

`preloadTime`

## preloadEventAbort

### Syntax

```
the preloadEventAbort
```

### Description

Movie property; specifies whether pressing keys or clicking the mouse can stop the preloading of cast members (TRUE) or not (FALSE, default).

This property can be tested and set. Setting this property affects the current movie.

### Example

This statement lets the user stop the preloading of cast members by pressing keys or clicking the mouse button:

```
set the preloadEventAbort = TRUE
```

### See also

`preLoad (command)`, `preLoadMember`

## preLoadMember

### Syntax

```
preLoadMember  
member(whichCastMember).preLoad()  
preLoadMember whichCastMember  
member(fromCastmember).preLoad(toCastMember)  
preLoadMember fromCastmember, toCastmember
```

### Description

Command; preloads cast members and stops when memory is full or when all of the specified cast members have been preloaded. The `preLoadMember` command returns the cast member number of the last cast member successfully loaded. To obtain this value, use the `result` function.

When used without arguments, `preLoadMember` preloads all cast members in the movie.

When used with the *whichCastMember* argument, `preLoadMember` preloads just that cast member. If *whichCastMember* is an integer, only the first cast is referenced. If *whichCastMember* is a string, the first member with the string as its name will be used.

When used with the arguments *fromCastmember* and *toCastmember*, the `preLoadMember` command preloads all cast members in the range specified by the cast member numbers or names.

### Examples

This statement preloads cast member 20 of the first (internal) cast:

```
member(20).preLoad()
```

This statement preloads cast member Shrine and the 10 cast members after it in the Cast window:

```
member("Shrine").preLoad(member("Shrine").number + 10)
```

To preload a specific member of a specific cast, use the following syntax:

```
member("John Boy", "Family Members").preLoad()
```

## preLoadMode

### Syntax

```
castLib(whichCast).preLoadMode  
the preLoadMode of castLib whichCast
```

### Description

Cast member property; determines the specified cast's preload mode and has the same effect as setting Load Cast in the Cast Properties dialog box. Possible values are the following:

- 0—Load cast when needed.
- 1—Load cast before frame 1.
- 2—Load cast after frame 1.

The default value for cast members is 0, when needed.

An on `prepareMovie` handler is usually a good place for Lingo that determines when cast members are loaded.

This property can be tested and set.

#### Example

The following statement tells Director to load the members of the cast Buttons before the movie enters frame 1:

```
CastLib("Buttons").preloadMode = 1
```

## preloadMovie

#### Syntax

```
preloadMovie whichMovie
```

#### Description

Command; preloads the data and cast members associated with the first frame of the specified movie. Preloading a movie helps it start faster when it is started by a `go to movie` or `play movie` command.

To preload cast members from a URL, use `preloadNetThing()` to load the cast members directly into the cache, or use `downloadNetThing` to load a movie on a local disk from which you can load the movie into memory and minimize downloading time.

#### Example

This statement preloads the movie Introduction, which is located in the same folder as the current movie:

```
preloadMovie "Introduction"
```

## preloadNetThing()

#### Syntax

```
preloadNetThing (url)
```

#### Description

Function; preloads a file from the Internet to the local cache so it can be used later without a download delay. Replace *url* with the name of any valid Internet file, such as a Director movie, graphic, or FTP server location. The return value is a network ID that you can use to monitor the progress of the operation.

The Director player for Java doesn't support this command because Java's security model doesn't allow writing to the local disk.

The `preloadNetThing()` function downloads the file while the current movie continues playing. Use `netDone()` to find out whether downloading is finished.

After an item is downloaded, it can be displayed immediately because it is taken from the local cache rather than from the network.

Although many network operations can be active at a time, running more than four concurrent operations usually slows down performance unacceptably.

Neither the cache size nor the Check Documents option in a browser's preferences affects the behavior of the `preloadNetThing` function.

The `preloadNetThing()` function does not parse a Director file's links. Thus, even if a Director file is linked to casts and graphic files, `preloadNetThing()` downloads only the Director file. You still must preload other linked objects separately.

### Example

This statement uses `preloadNetThing()` and returns the network ID for the operation:

```
set mynetid = preloadNetThing("http://www.yourserver.com/menupage/  
mymovie.dir")
```

After downloading is complete, you can navigate to the movie using the same URL. The movie will be played from the cache instead of the URL, since it's been loaded in the cache.

### See also

`netDone()`

## preloadRAM

### Syntax

the `preloadRAM`

### Description

System property; specifies the amount of RAM that can be used for preloading a digital video. This property can be set and tested.

This property is useful for managing memory, limiting digital video cast members to a certain amount of memory, so that other types of cast members can still be preloaded. When `preloadRAM` is `FALSE`, all available memory can be used for preloading digital video cast members.

However, it's not possible to reliably predict how much RAM a digital video will require once it is preloaded, because memory requirements are affected by the content of the movie, how much compression was performed, the number of keyframes, changing imagery, and so on.

It is usually safe to preload the first couple of seconds of a video and then continue streaming from that point on.

If you know the data rate of your movie, you can estimate the setting for `preloadRAM`. For example, if your movie has a data rate of 300K per second, set `preloadRAM` to 600K if you want to preload the first 2 seconds of the video file. This is only an estimate, but it works in most situations.

### Example

This statement sets `preloadRAM` to 600K, to preload the first 2 seconds of a movie with a data rate of 300K per second:

```
set the preloadRAM = 600
```

### See also

`loop` (keyword), `next`

## preloadTime

### Syntax

```
member(whichCastMember).preloadTime  
the preloadTime of member whichCastMember  
sound(channelNum).preloadTime
```

### Description

Cast member and sound channel property; for cast members, specifies the amount of the Shockwave Audio (SWA) streaming cast member to download, in seconds, before playback begins or when a `preloadBuffer` command is used. The default value is 5 seconds.

This property can be set only when the SWA streaming cast member is stopped.

For sound channels, the value is for the given sound in the queue or the currently playing sound if none is specified.

### Examples

The following handler sets the preload download time for the SWA streaming cast member Louis Armstrong to 6 seconds. The actual preload occurs when a `preLoadBuffer` or `play` command is issued.

```
on mouseDown
  member("Louis Armstrong").stop()
  member("Louis Armstrong").preLoadTime = 6
end
```

This statement returns the `preLoadTime` of the currently playing sound in sound channel 1:

```
put sound(1).preLoadTime
```

### See also

`preLoadBuffer` `member`

## preMultiply

### Syntax

```
transform1.preMultiply(transform2)
```

### Description

3D transform command; alters *transform1* by pre-applying the positional, rotational, and scaling effects of *transform2*.

If *transform2* describes a rotation of 90° about the X axis and *transform1* describes a translation of 100 units in the Y axis, `transform1.multiply\(transform2)` will alter this transform so that it describes a translation followed by a rotation. The statement `transform1.preMultiply(transform2)` will alter this transform so that it describes a rotation followed by a translation. The effect is that the order of operations is reversed.

### Example

This statement performs a calculation that applies the transform of the model Mars to the transform of the model Pluto:

```
member("scene").model("Pluto").transform.preMultiply\
  (member("scene").model("Mars").transform)
```

## on prepareFrame

### Syntax

```
on prepareFrame
  statement(s)
end
```

### Description

System message and event handler; contains statements that run immediately before the current frame is drawn.

Unlike `beginSprite` and `endSprite` events, a `prepareFrame` event is generated each time the playhead enters a frame.



The `on prepareFrame` handler is a useful place to change sprite properties before the sprite is drawn.

If used in a behavior, the `on prepareFrame` handler receives the reference `me`.

The `go`, `play`, and `updateStage` commands are disabled in an `on prepareFrame` handler.

#### Example

This handler sets the `loch` property of the sprite that the behavior is attached to:

```
on prepareFrame me
    sprite(me.spriteNum).loch = the mouseH
end
```

#### See also

`on enterFrame`

## on prepareMovie

#### Syntax

```
on prepareMovie
    statement(s)
end
```

#### Description

System message and event handler; contains statements that run after the movie preloads cast members but before the movie does the following:

- Creates instances of behaviors attached to sprites in the first frame that plays.
- Prepares the first frame that plays, including drawing the frame, playing any sounds, and executing transitions and palette effects.

New global variables used for sprite behaviors in the first frame should be initialized in the `on prepareMovie` handler. Global variables already set by the previous movie do not need to be reset.

An `on prepareMovie` handler is a good place to put Lingo that creates global variables, initializes variables, plays a sound while the rest of the movie is loading into memory, or checks and adjusts computer conditions such as color depth.

The `go`, `play`, and `updateStage` commands are disabled in an `on prepareMovie` handler.

#### Example

This handler creates a global variable when the movie starts:

```
on prepareMovie
    global currentScore
    set currentScore = 0
end
```

#### See also

`on enterFrame`, `on startMovie`

# preRotate

## Syntax

```
transformReference.preRotate( xAngle, yAngle, zAngle )
transformReference.preRotate( vector )
transformReference.preRotate( positionVector, directionVector, \
    angle )
member( whichCastmember ).node.transform.preRotate( xAngle, \
    yAngle, zAngle )
member( whichCastmember ).node.transform.preRotate( vector )
member( whichCastmember ).node.transform.preRotate(
    ( positionVector, directionVector, angle )
```

## Description

3D transform command; applies a rotation before the current positional, rotational, and scale offsets held by the referenced transform object. The rotation may be specified as a set of three angles, each of which specify an angle of rotation about the three corresponding axes. These angles may be specified explicitly in the form of *xAngle*, *yAngle*, and *zAngle*, or by a vector, where the x component of the vector corresponds to the rotation about the x-axis, the y about the y-axis, and the z about the z-axis.

Alternatively, the rotation may also be specified as a rotation about an arbitrary axis. This axis is defined in space by *positionVector* and *directionVector*. The amount of rotation about this axis is specified by *angle*.

*Node* may be a reference to a model, group, light, or camera

## Example

The following statement performs a rotation of 20° about each axis. Since the model's *transform* property is its position, rotation, and scale offsets relative to that model's parent, and *preRotate* applies the change in orientation prior to any existing effects of that model's transform, this will rotate the model in place rather than orbiting around its parent.

```
member("scene").model("bip01").transform.preRotate(20, 20, 20)
```

Note that the above is equivalent to:

```
member("scene").model("bip01").rotate(20,20,20).
```

Generally *preRotate()* is only useful when dealing with transform variables. This line will orbit the camera about the point (100, 0, 0) in space, around the y axis, by 180°.

```
t = transform()
t.position = member("scene").camera[1].transform.position
t.preRotate(vector(100, 0, 0), vector(0, 1, 0), 180)
member("scene").camera[1].transform = t
```

## See also

[rotate](#)

## previous

### See

[go previous](#)

## preScale()

### Syntax

```
transformReference.preScale( xScale, yScale, zScale )
transformReference.preScale( vector )
member( whichCastmember ).node.transform.preScale( xScale, \
yScale, zScale )
member( whichCastmember ).node.transform.preScale( vector )
```

### Description

3D transform command; applies a scale prior to the existing positional, rotational, and scaling effects of the given transform.

*Node* may be a reference to a model, group, light, or camera.

### Example

**Line 1** of the following Lingo creates a duplicate of Moon1's transform. Remember that access to a model's transform property is by reference.

**Line 2** applies a scale to that transform prior to any existing positional or rotational effects of that transform. Assume that the transform represents the positional offset and rotational orbit of Moon1 relative to its parent planet. Lets also assume Moon2's parent is the same as Moon1's. If we used `scale()` here instead of `preScale()`, then Moon2 would be pushed out twice as far and rotated about the planet twice as much as is Moon1. This is because the scaling would be applied to the transform's existing positional and rotational offsets. Using `preScale()` will apply the size change without affecting these existing positional and rotational offsets.

**Line 3** applies an additional 180° rotation about the *x*-axis of the planet. This will put Moon2 on the opposite side of Moon1's orbit. Using `preRotate()` would have left Moon2 in the same place as Moon1, spun around its own *x*-axis by 180°.

**Line 4** assigns this new transform to Moon2.

```
t = member("scene").model("Moon1").transform.duplicate()
t.preScale(2,2,2)
t.rotate(180,0,0)
member("scene").model("Moon2").transform = t
```

## preTranslate()

### Syntax

```
transformReference.preTranslate( xIncrement, yIncrement, \
zIncrement )
transformReference.preTranslate( vector )
member( whichCastmember ).node.transform.preTranslate(
xIncrement, yIncrement, zIncrement )
member( whichCastmember ).node.transform.preTranslate( vector )
```

### Description

3D transform command; applies a translation before the current positional, rotational, and scale offsets held by the referenced transform object. The translation may be specified as a set of three increments along the three corresponding axes. These increments may be specified explicitly in the form of *xIncrement*, *yIncrement*, and *zIncrement*, or by a vector, where the X component of the vector corresponds to the translation about the X axis, the Y about the Y axis, and the Z about the Z axis.

After a series of transformations are done, in the following order, the model's local origin will be at (0, 0, -100), assuming the model's parent is the world:

```
model.transform.identity()
model.transform.rotate(0, 90, 0)
model.transform.preTranslate(100, 0, 0)
```

Had `translate()` been used instead of `preTranslate()`, the model's local origin would be at (100, 0, 0) and the model rotated about its own Y axis by 90°. Note that

`model.transform.pretranslate(x, y, z)` is equivalent to `model.translate(x, y, z)`.

Generally, `preTranslate()` is only useful when dealing with transform variables rather than `model.transform` references.

#### Example

```
t = transform()
t.transform.identity()
t.transform.rotate(0, 90, 0)
t.transform.preTranslate(100, 0, 0)
gbModel = member("scene").model("mars")
gbModel.transform = t
put gbModel.transform.position
-- vector(0.0000, 0.0000, -100.0000)
```

## primitives

#### Syntax

```
getRendererServices().primitives
```

#### Description

3D function; returns a list of the primitive types that can be used to create new model resources.

#### Example

This statement display the available primitive types:

```
put getRendererServices().primitives
-- [#sphere, #box, #cylinder, #plane, #particle]
```

#### See also

```
getRendererServices(), newModelResource
```

## print()

### Syntax

```
sprite(whichSprite).print({"targetName", #printingBounds})
```

### Description

Command; calls the corresponding `print` ActionScript command, which was introduced in Flash 5. All frames in the Flash movie that have been labeled `#p` are printed. If no individual frames have been labeled, the whole movie prints.

Both arguments to this function are optional. The target movie is the movie or movie clip to be printed. If you do not specify a target (or if the target is 0), then the main Flash movie is printed.

The two options for the printing bounds are `#bframe` and `#bmax`. If `#bmax` is specified, then the printing bounds become a large enough virtual rectangle to fit all frames to be printed. If `#bframe` is specified, then the printing bounds for each page are changed to match each frame that is being printed. If no printing bounds are specified, the bounds of the target movie are used.

Because printing of Flash movies is rather complicated, you may benefit from reviewing the section about printing in the Flash 5 documentation before using this sprite function.

## printAsBitmap()

### Syntax

```
sprite(whichSprite).printAsBitmap({"targetName", #printingBounds})
```

### Description

Command; functions much like the `print` command. However, `printAsBitmap` can be used to print objects containing alpha channel information.

## printFrom

### Syntax

```
printFrom fromFrame {,toFrame} {,reduction}
```

### Description

Command; prints whatever is displayed on the Stage in each frame, whether or not the frame is selected, starting at the frame specified by *fromFrame*. Optionally, you can supply *toFrame* and a reduction value (100%, 50%, or 25%).

The frame being printed need not be currently displayed. This command always prints at 72 dots per inch (dpi), bitmaps everything on the screen (text will not be as smooth in some cases), prints in portrait (vertical) orientation, and ignores Page Setup settings. For more flexibility when printing from within Director, see PrintOMatic Lite Xtra, which is on the installation disk.

### Examples

This statement prints what is on the Stage in frame 1:

```
printFrom 1
```

The following statement prints what is on the Stage in every frame from the marker Intro to the marker Tale. The reduction is 50%.

```
printFrom label("Intro"), label("Tale"), 50
```

## projection

### Syntax

```
sprite(whichSprite).camera.projection  
camera(whichCamera).projection  
member(whichCastmember).camera(whichCamera).projection
```

### Description

3D property; allows you to get or set the projection style of the camera. Possible values are `#perspective` (the default) and `#orthographic`.

When projection is `#perspective`, objects closer to the camera appear larger than objects farther from the camera, and the `projectionAngle` or `fieldOfView` properties specify the vertical projection angle (which determines how much of the world you see). The horizontal projection angle is determined by the aspect ratio of the camera's `rect` property.

When projection is `#orthographic`, the apparent size of objects does not depend on distance from the camera, and the `orthoHeight` property specifies how many world units fit vertically into the sprite (which determines how much of the world you see). The orthographic projection width is determined by the aspect ratio of the camera's `rect` property.

### Example

This statement sets the projection property of the camera of sprite 5 to `#orthographic`:

```
sprite(5).camera.projection = #orthographic
```

### See also

`fieldOfView` (3D), `orthoHeight`, `projectionAngle`

## projectionAngle

This Lingo is obsolete. Use `fieldOfView` instead.

### See also

`fieldOfView` (3D)

# property

## Syntax

```
property {property1}{, property2} {,property3} {...}
```

## Description

**Keyword;** declares the properties specified by *property1*, *property2*, and so on as property variables.

Declare property variables at the beginning of the parent script or behavior script. You can access them from outside the parent script or behavior script by using the `the` operator.

**Note:** The `spriteNum` property is available to all behaviors and simply needs to be declared to be accessed.

You can refer to a property within a parent script or behavior script without using the `me` keyword. However, to refer to a property of a parent script's ancestor, use the form `me.property`.

For behaviors, properties defined in one behavior script are available to other behaviors attached to the same sprite.

You can directly manipulate a child object's property from outside the object's parent scripts through syntax similar to that for manipulating other properties. For example, this statement sets the `motionStyle` property of a child object:

```
set the motionStyle of myBouncingObject to #frenetic
```

Use the `count` function to determine the number of properties within the parent script of a child object. Retrieve the name of these properties by using `getPropAt`. Add properties to an object by using `setaProp()`.

To see an example of `property` used in a completed movie, see the Parent Scripts movie in the Learning/Lingo Examples folder inside the Director application folder.

## Examples

This statement lets each child object created from a single parent script have its own location and velocity setting:

```
property location, velocity
```

This parent script handler declares `pMySpriteNum` a property to make it available:

```
-- script Elder
property pMyChannel

on new me, whichSprite
    me.pMyChannel = whichSprite
    return me
end
```

The original behavior script sets up the ancestor and passes the `spriteNum` property to all behaviors:

```
property spriteNum
property ancestor

on beginSprite me
    set ancestor = new(script "Elder", spriteNum)
end
```

## See also

`me`, `ancestor`, `spriteNum`

## proxyServer

### Syntax

```
proxyServer serverType, "ipAddress", portNum  
proxyServer()
```

### Description

Command; sets the values of an FTP or HTTP proxy server, as follows:

- *serverType*—#ftp or #http
- *ipAddress*—A string containing the IP address
- *portNum*—The integer value of the port number

If you use the syntax `proxyServer()`, this element returns the settings of an FTP or HTTP proxy server.

### Examples

This statement sets up an HTTP proxy server at IP address 197.65.208.157 using port 5:

```
proxyServer #http,"197.65.208.157",5
```

This statement returns the port number of an HTTP proxy server:

```
put proxyServer(#http,#port)
```

If no server type is specified, the function returns 1.

This statement returns the IP address string of an HTTP proxy server:

```
put proxyServer(#http)
```

This statement turns off an FTP proxy server:

```
proxyServer #ftp,#stop
```

## ptToHotSpotID()

### Syntax

```
ptToHotSpotID(whichQTVRSprite, point)
```

### Description

QuickTime VR function; returns the ID of the hotspot, if any, that is at the specified point. If there is no hotspot, the function returns 0.

## puppet

### Syntax

```
sprite(whichSprite).puppet  
the puppet of sprite whichSprite
```

### Description

Sprite property; determines whether the sprite channel specified by *whichSprite* is a puppet under Lingo control (TRUE) or not (FALSE, default).

- If a sprite channel is a puppet, any changes that Lingo makes to the channel's sprite properties remain in effect after the playhead leaves the sprite.
- If a sprite channel is not a puppet, any changes that Lingo makes to a sprite last for the life of the current sprite only.



While the playhead is in the same sprite, setting the sprite channel's `puppet sprite` property to `FALSE` resets the sprite's properties to those set in the Score.

Making the sprite channel a puppet lets you control many sprite properties, such as `member`, `locH`, and `width`, from Lingo after the playhead exits from the sprite.

Setting the `puppet sprite` property is equivalent to using the `puppetSprite` command. For example, the following statements are equivalent: set the puppet of sprite 1 to `TRUE` and `puppetSprite 1, TRUE`.

This property can be tested and set.

### Examples

This statement makes the sprite numbered `i + 1` a puppet:

```
sprite(i + 1).puppet = TRUE
```

The following statement records whether sprite 5 is a puppet by assigning the value of the `puppet sprite` property to the variable. When sprite 5 is a puppet, `isPuppet` is set to `TRUE`. When sprite 5 is not a puppet, `isPuppet` is set to `FALSE`.

```
isPuppet = sprite(5).puppet
```

### See also

`puppetSprite`

## puppetPalette

### Syntax

```
puppetPalette whichPalette {, speed} {,nFrames}
```

### Description

Command; causes the palette channel to act as a puppet and lets Lingo override the palette setting in the palette channel of the Score and assign palettes to the movie.

The `puppetPalette` command sets the current palette to the palette cast member specified by *whichPalette*. If *whichPalette* evaluates to a string, it specifies the cast name of the palette. If *whichPalette* evaluates to an integer, it specifies the member number of the palette.

For best results, use the `puppetPalette` command before navigating to the frame on which the effect will occur so that Director can map to the desired palette before drawing the next frame.

You can fade in the palette by replacing *speed* with an integer from 1 (slowest) to 60 (fastest). You can also fade in the palette over several frames by replacing *nFrames* with an integer for the number of frames.

A puppet palette remains in effect until you turn it off with the command `puppetPalette 0`. No subsequent palette changes in the Score are obeyed when the puppet palette is in effect.

**Note:** The browser controls the palette for the entire Web page. Thus, Shockwave and the Director player for Java always uses the browser's palette.

For the most reliable color when authoring a movie for playback as a Director player for Java, use the default palette for the authoring system.

## Examples

This statement makes Rainbow the movie's palette:

```
puppetPalette "Rainbow"
```

The following statement makes Grayscale the movie's palette. The transition to the Grayscale palette occurs over a time setting of 15 and between frames labeled Gray and Color.

```
puppetPalette "Grayscale", 15, label("Gray") - label("Color")
```

## puppetSound

### Syntax

```
puppetSound whichChannel, whichCastMember  
puppetSound whichCastMember  
puppetSound member whichCastMember  
puppetSound 0  
puppetSound whichChannel, 0
```

### Description

Command; makes the sound channel a puppet, plays the sound cast member specified by *whichCastMember*, and lets Lingo override any sounds assigned in the Score's sound channels.

Specify a sound channel by replacing *whichChannel* with a channel number.

The sound starts playing after the playhead moves or the `updateStage` command is executed. Using 0 as the cast number argument stops the sound from playing. It also returns control of the sound channel to the Score.

Puppet sounds can be useful for playing a sound while a different movie is being loaded into memory.

The Director player for Java supports the following versions of the `puppetSound` command:

- `puppetSound whichChannel, whichCastMember`, or `puppetSound whichCastMember—Plays a sound.`
- `puppetSound 0` or `puppetSound whichChannel, 0`—Stops a sound.

### Examples

This statement plays the sound Wind under control of Lingo:

```
puppetSound "Wind"
```

This statement turns off the sound playing in channel 2:

```
puppetSound 2, 0
```

### See also

`sound fadeIn`, `sound fadeOut`, `sound playFile`, `sound stop`

# puppetSprite

## Syntax

`puppetSprite whichChannel, state`

## Description

Command; determines whether the sprite channel specified by *whichSprite* is a puppet and under Lingo control (TRUE) or not a puppet and under the control of the Score (FALSE).

While the playhead is in the same sprite, turning off the sprite channel's puppetting using the command `puppetSprite whichSprite, FALSE` resets the sprite's properties to those in the Score.

The sprite channel's initial properties are whatever the channel's settings are when the `puppetSprite` command is executed. You can use Lingo to change sprite properties as follows:

- If a sprite channel is a puppet, any changes that Lingo makes to the channel's sprite properties remain in effect after the playhead exits the sprite.
- If a sprite channel is not a puppet, any changes that Lingo makes to a sprite last for the life of the current sprite only.

The channel must contain a sprite when you use the `puppetSprite` command.

Making the sprite channel a puppet lets you control many sprite properties—such as `memberNum`, `locH`, and `width`—from Lingo after the playhead exits the sprite.

Use the command `puppetSprite whichSprite, FALSE` to return control to the Score when you finish controlling a sprite channel from Lingo and to avoid unpredictable results that may occur when the playhead is in frames that aren't intended to be puppets.

**Note:** Version 6 of Director introduced autopuppetting, which made it unnecessary to explicitly puppet a sprite under most circumstances. Explicit control is still useful if you want to retain complete control over a channel's contents even after a sprite span has finished playing.

## Examples

This statement makes the sprite in channel 15 a puppet:

```
puppetSprite 15, TRUE
```

This statement removes the puppet condition from the sprite in the channel numbered `i + 1`:

```
puppetSprite i + 1, FALSE
```

## See also

`backColor`, `bottom`, `constraint`, `cursor` (command), `foreColor`, `height`, `ink`, `left`, `lineSize`, `locH`, `locV`, `memberNum`, `puppet`, `right`, `top`, `type` (sprite property), `width`

# puppetTempo

## Syntax

`puppetTempo framesPerSecond`

## Description

Command; causes the tempo channel to act as a puppet and sets the tempo to the number of frames specified by *framesPerSecond*. When the tempo channel is a puppet, Lingo can override the tempo setting in the Score and change the tempo assigned to the movie.

It's unnecessary to turn off the puppet tempo condition to make subsequent tempo changes in the Score take effect.

**Note:** Although it is theoretically possible to achieve frame rates up to 30,000 frames per second (fps) with the `puppetTempo` command, you could do this only with little animation and a very powerful machine.

**Examples**

This statement sets the movie’s tempo to 30 fps:

```
puppetTempo 30
```

This statement increases the movie’s old tempo by 10 fps:

```
puppetTempo oldTempo + 10
```

**puppetTransition**

**Syntax**

```
puppetTransition member whichCastMember  
puppetTransition whichTransition {,time} {, chunkSize} {, changeArea}
```

**Description**

Command; performs the specified transition between the current frame and the next frame.

To use an Xtra transition cast member, use `puppetTransition member` followed by the cast member’s name or number.

To use a built-in Director transition, replace *whichTransition* with a value in the following table. Replace *time* with the number of quarter seconds used to complete the transition. The minimum value is 0; the maximum is 120 (30 seconds). Replace *chunkSize* with the number of pixels in each chunk of the transition. The minimum value is 1; the maximum is 128. Smaller chunk sizes yield smoother transitions but are slower.

Code	Transition	Code	Transition
01	Wipe right	27	Random rows
02	Wipe left	28	Random columns
03	Wipe down	29	Cover down
04	Wipe up	30	Cover down, left
05	Center out, horizontal	31	Cover down, right
06	Edges in, horizontal	32	Cover left
07	Center out, vertical	33	Cover right
08	Edges in, vertical	34	Cover up
09	Center out, square	35	Cover up, left
10	Edges in, square	36	Cover up, right
11	Push left	37	Venetian blinds
12	Push right	38	Checkerboard
13	Push down	39	Strips on bottom, build left
14	Push up	40	Strips on bottom, build right
15	Reveal up	41	Strips on left, build down
16	Reveal up, right	42	Strips on left, build up
17	Reveal right	43	Strips on right, build down
18	Reveal down, right	44	Strips on right, build up

19	Reveal down	45	Strips on top, build left
20	Reveal down, left	46	Strips on top, build right
21	Reveal left	47	Zoom open
22	Reveal up, left	48	Zoom close
23	Dissolve, pixels fast*	49	Vertical blinds
24	Dissolve, boxy rectangles	50	Dissolve, bits fast*
25	Dissolve, boxy squares	51	Dissolve, pixels*
26	Dissolve, patterns	52	Dissolve, bits*

Transitions marked with an asterisk (\*) do not work on monitors set to 32 bits.

There is no direct relationship between a low time value and a fast transition. The actual speed of the transition depends on the relation of *chunkSize* and *time*. For example, if *chunkSize* is 1 pixel, the transition takes longer no matter how low the time value, because the computer has to do a lot of work. To make transitions occur faster, use a larger chunk size, not a shorter time.

Replace *changeArea* with a value that determines whether the transition occurs only in the changing area (TRUE) or over the entire Stage (FALSE, default). The *changeArea* variable is an area within which sprites have changed.

#### Example

The following statement performs a wipe right transition. Because no value is specified for *changeArea*, the transition occurs over the entire Stage, which is the default.

```
puppetTransition 1
```

This statement performs a wipe left transition that lasts 1 second, has a chunk size of 20, and occurs over the entire Stage:

```
puppetTransition 2, 4, 20, FALSE
```

## purgePriority

### Syntax

```
member(whichCastMember).purgePriority  
the purgePriority of member whichCastMember
```

### Description

Cast member property; specifies the purge priority of the cast member specified by *whichCastMember*.

Cast members' purge priorities determine the priority that Director follows to choose which cast members to delete from memory when memory is full. The higher the purge priority, the more likely that the cast member will be deleted. The following *purgePriority* settings are available:

- 0—Never
- 1—Last
- 2—Next
- 3—Normal

Normal, which is the default, lets Director purge cast members from memory at random. Next, Last, and Never allow some control over purging, but Last or Never may cause your movie to run out of memory if several cast members are set to these values.

Setting `purgePriority` for cast members is useful for managing memory when the size of the movie's cast exceeds the available memory. As a general rule, you can minimize pauses while the movie loads cast members and reduce the number of times Director reloads a cast member by assigning a low purge priority to cast members that are used frequently in the course of the movie.

#### Example

This statement sets the purge priority of cast member Background to 3, which makes it one of the first cast members to be purged when memory is needed:

```
member("Background").purgePriority = 3
```

## put

#### Syntax

```
put expression
```

#### Description

Command; evaluates the expression specified by *expression* and displays the result in the Message window. This command can be used as a debugging tool by tracking the values of variables as a movie plays.

The Director player for Java displays the message from the `put` command in the browser's Java console window. Access to the console window depends on the browser.

#### Examples

This statement displays the time in the Message window:

```
put the time
-- "9:10 AM"
```

This statement displays the value assigned to the variable `bid` in the Message window:

```
put bid
-- "Johnson"
```

#### See also

`put...after`, `put...before`, `put...into`

## put...after

#### Syntax

```
put expression after chunkExpression
```

#### Description

Command; evaluates a Lingo expression, converts the value to a string, and inserts the resulting string after a specified chunk in a container, without replacing the container's contents. (If *chunkExpression* specifies a nonexistent target chunk, the string value is inserted as appropriate into the container.)

Chunk expressions refer to any character, word, item, or line in any container. Containers include field cast members; text cast members; variables that hold strings; and specified characters, words, items, lines, and ranges within containers.

### Examples

This statement adds the string “fox dog cat” after the contents of the field cast member `Animal List`:

```
put "fox dog cat" after member "Animal List"
```

The same can be accomplished using this statement:

```
put "fox dog cat" after member("Animal List").line[1]
```

### See also

`char...of`, `item...of`, `line...of`, `paragraph`, `word...of`, `put...before`, `put...into`

## put...before

### Syntax

```
put expression before chunkExpression
```

### Description

Command; evaluates a Lingo expression, converts the value to a string, and inserts the resulting string before a specified chunk in a container, without replacing the container's contents. (If *chunkExpression* specifies a nonexistent target chunk, the string value is inserted as appropriate into the container.)

Chunk expressions refer to any character, word, item, or line in any container. Containers include field cast members; text cast members; variables that hold strings; and specified characters, words, items, lines, and ranges in containers.

### Examples

This statement sets the variable `animalList` to the string “fox dog cat” and then inserts the word *elk* before the second word of the list:

```
put "fox dog cat" into animalList
put "elk " before word 2 of animalList
```

The result is the string “fox elk dog cat”.

The same can be accomplished using this syntax:

```
put "fox dog cat" into animalList
put "elk " before animalList.word[2]
```

### See also

`char...of`, `item...of`, `line...of`, `paragraph`, `word...of`, `put...after`, `put...into`

## put...into

### Syntax

```
put expression into chunkExpression
```

### Description

Command; evaluates a Lingo expression, converts the value to a string, and uses the resulting string to replace a specified chunk in a container. (If *chunkExpression* specifies a nonexistent target chunk, the string value is inserted as appropriate into the container.)

Chunk expressions refer to any character, word, item, or line in any container. Containers include field cast members; text cast members; variables that hold strings; and specified characters, words, items, lines, and ranges in containers.

When a movie plays back as an applet, the `put...into` command replaces all text within a container, not chunks of text.

To assign values to variables, use the `set` command.

### Examples

This statement changes the second line of the field cast member Review Comments to “Reviewed by Agnes Gooch”:

```
put "Reviewed by Agnes Gooch" into line 2 of member "Review Comments"
```

The same can be accomplished with a text cast member using this syntax:

```
put "Reviewed by Agnes Gooch" into member("Review Comments").line[2]
```

### See also

`char...of`, `item...of`, `line...of`, `paragraph`, `word...of`, `put...before`, `put...after`, `set...to`, `set...=`



## qtRegisterAccessKey

### Syntax

```
qtRegisterAccessKey(categoryString, keyString)
```

### Description

Command; allows registration of a key for encrypted QuickTime media.

The key is an application-level key, not a system-level key. After the application unregisters the key or shuts down, the media will no longer be accessible.

**Note:** For security reasons, there is no way to display a listing of all registered keys.

### See also

qtUnRegisterAccessKey

## qtUnRegisterAccessKey

### Syntax

```
qtUnRegisterAccessKey(categoryString, keyString)
```

### Description

Command; allows the key for encrypted QuickTime media to be unregistered.

The key is an application-level key, not a system-level key. After the application unregisters the key, only movies encrypted with this key continue to play. Other media will no longer be accessible.

### See also

qtRegisterAccessKey

## quad

### Syntax

```
sprite(whichSpriteNumber).quad
```

### Description

Sprite property; contains a list of four points, which are floating point values that describe the corner points of a sprite on the Stage. The points are organized in the following order: upper left, upper right, lower right, and lower left.

The points themselves can be manipulated to create perspective and other image distortions.

After you manipulate the quad of a sprite, you can reset it to the Score values by turning off the puppet of the sprite with `puppetSprite whichSpriteNumber, FALSE`. When the quad of a sprite is disabled, you cannot rotate or skew the sprite.

## Examples

This statement displays a typical list describing a sprite:

```
put sprite(1).quad
-- [point(153.0000, 127.0000), point(231.0000, 127.0000), \
   point(231.0000, 242.0000), point(153.0000, 242.0000)]
```

When modifying the `quad` sprite property, be aware that you must reset the list of points after changing any of the values. This is because when you set a variable to the value of a property, you are placing a copy of the list, not the list itself, in the variable. To effect a change, use syntax like this:

```
-- Get the current property contents
currQuadList = sprite(5).quad
-- Add 50 pixels to the horizontal and vertical positions of the first point in the list
currQuadList[1] = currQuadList[1] + point(50, 50)
-- Reset the actual property to the newly computed position
sprite(5).quad = currQuadList
```

## See also

`rotation`, `skew`

# quality

## Syntax

```
sprite(whichFlashSprite).quality
the quality of sprite whichFlashSprite
member(whichFlashMember).quality
the quality of member whichFlashMember
```

## Description

Flash cast member and sprite property; controls whether Director uses anti-aliasing to render a Flash movie sprite, producing high-quality rendering but possibly slower movie playback. The `quality` property can have these values:

- `#autoHigh`—Director starts by rendering the sprite with anti-aliasing. If the actual frame rate falls below the movie's specified frame rate, Director turns off anti-aliasing. This setting gives precedence to playback speed over visual quality.
- `#autoLow`—Director starts by rendering the movie without anti-aliasing. If the Flash player determines that the computer processor can handle it, anti-aliasing is turned on. This setting gives precedence to visual quality whenever possible.
- `#high` (default)—The movie always plays with anti-aliasing.
- `#low`—The movie always plays without anti-aliasing.

The `quality` property can be tested and set.

## Example

The following sprite script checks the color depth of the computer on which the movie is playing. If the color depth is set to 8 bits or less (256 colors), the script sets the quality of the sprite in channel 5 to `#low`.

```
on beginSprite me
  if the colorDepth <= 8 then
    sprite(1).quality = #low
  end if
end
```

## quality (3D)

### Syntax

```
member(whichCastmember).texture(whichTexture).quality
member(whichCastmember).shader(whichShader).texture\
(whichTexture).quality
member(whichCastmember).model(whichModel).shader.texture\
(whichTexture).quality
member(whichCastmember).model(whichModel).\
  shader.texturelist[TextureListIndex].quality
member(whichCastmember).model(whichModel).shaderList\
[shaderListIndex].texture(whichTexture).quality
member(whichCastmember).model(whichModel).shaderList\
[shaderListIndex].texturelist[TextureListIndex].quality
```

### Description

3D texture property; lets you get or set the image quality of a texture by controlling the level of mipmapping applied to the texture. Mipmapping is a process by which additional versions of the texture image are created in several sizes that are smaller than the original image. The 3D Xtra then uses whichever version of the image is most appropriate to the current size of the model on the screen and changes the version of the image that is being used when needed. Trilinear mipmapping is higher in quality and uses more memory than bilinear mipmapping.

Mipmapping is not the same as filtering, although both improve texture appearance. Filtering spreads errors out across the texture's area so that errors are less concentrated. Mipmapping resamples the image to make it the appropriate size.

This property can have the following values:

- `#low` is the same as `off`, and mipmapping is not used for the texture.
- `#medium` enables a low-quality (bilinear) mipmapping for the texture.
- `#high` enables a high-quality (trilinear) mipmapping for the texture.

The default is `#low`.

### Example

This statement sets the `quality` property of the texture `Marsmap` to `#medium`:

```
member("scene").texture("Marsmap").quality = #medium
```

### See also

`nearFiltering`

## queue()

### Syntax

```
sound(channelNum).queue(member(whichMember))
sound(channelNum).queue([#member member(whichMember),#startTime:milliseconds#endTime:
  milliseconds,#loopCount:numberOfLoops,#loopStartTime:milliseconds,#loopEndTime:
  milliseconds,#preloadTime: milliseconds])
queue(sound(channelNum), member(whichMember))
queue(soundObject,[#member member(whichMember),{#startTime:milliseconds#endTime:
  milliseconds,#loopCount:numberOfLoops,#loopStartTime:milliseconds,#loopEndTime:
  milliseconds,#preloadTime: milliseconds}])
```

## Description

Function; adds the given sound cast member to the queue of sound channel *channelNum*.

Once a sound has been queued, it can be played immediately with the `play()` command. This is because Director preloads a certain amount of each sound that is queued, preventing any delay between the `play()` command and the start of playback. The default amount of sound that is preloaded is 1500 milliseconds. This parameter can be modified by passing a property list containing one or more parameters with the `queue()` command.

You can specify these properties may be specified with the `queue()` command:

Property	Description
<code>#member</code>	The sound cast member to queue. This property must be provided; all others are optional.
<code>#startTime</code>	The time within the sound at which playback begins, in milliseconds. The default is the beginning of the sound. See <code>startTime</code> .
<code>#endTime</code>	The time within the sound at which playback ends, in milliseconds. The default is the end of the sound. See <code>endTime</code> .
<code>#loopCount</code>	The number of times to play a loop defined with <code>#loopStartTime</code> and <code>#loopEndTime</code> . The default is 1. See <code>loopCount</code> .
<code>#loopStartTime</code>	The time within the sound to begin a loop, in milliseconds. See <code>loopStartTime</code> .
<code>#loopEndTime</code>	The time within the sound to end a loop, in milliseconds. See <code>loopEndTime</code> .
<code>#preloadTime</code>	The amount of the sound to buffer before playback, in milliseconds. See <code>preloadTime</code> .

These parameters can also be passed with the `setPlayList()` command.

To see an example of `queue()` used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

## Example

The following handler queues and plays two sounds. The first sound, cast member Chimes, is played in its entirety. The second sound, cast member introMusic, is played starting at its 3-second point, with a loop repeated 5 times from the 8-second point to the 8.9 second point, and stopping at the 10-second point.

```
on playMusic
  sound(2).queue([#member: member("Chimes")])
  sound(2).queue([#member: member("introMusic"), #startTime: 3000,\
    #endTime: 10000, #loopCount: 5, #loopStartTime: 8000, #loopEndTime: 8900])
  sound(2).play()
end
```

## See also

`startTime`, `endTime`, `loopCount`, `loopStartTime`, `loopEndTime`, `preloadTime`, `setPlayList()`, `play()` (sound)

## queue() (3D)

### Syntax

```
member(whichCastmember).model(whichModel).bonesPlayer.queue\  
    (motionName {, looped, startTime, endTime, scale, offset})  
member(whichCastmember).model(whichModel).keyframePlayer.\  
    queue(motionName {, looped, startTime, endTime, scale, offset})
```

### Description

3D keyframePlayer and bonesPlayer modifier command; adds the motions specified by *motionName* to the end of the modifier's *playlist* property. The motion is executed by the model when all the motions ahead of it in the playlist are finished playing.

The optional parameters of this command are as follows:

*looped* specifies whether the motion plays once (FALSE) or continuously (TRUE).

*startTime* is measured in milliseconds from the beginning of the motion. When *looped* is FALSE, the motion begins at *offset* and ends at *endTime*. When *looped* is TRUE, the first iteration of the loop begins at *offset* and ends at *endTime*. All subsequent repetitions begin at *startTime* and end at *endTime*.

*endTime* is measured in milliseconds from the beginning of the motion. When *looped* is FALSE, the motion begins at *offset* and ends at *endTime*. When *looped* is TRUE, the first iteration of the loop begins at *offset* and ends at *endTime*. All subsequent repetitions begin at *cropStart* and end at *endTime*. Set *endTime* to -1 if you want the motion to play to the end.

*scale* is multiplied by the *playRate* property of the model's keyframePlayer modifier or bonesPlayer modifier to determine the actual speed of the motion's playback.

*offset* is measured in milliseconds from the beginning of the motion. When *looped* is FALSE, the motion begins at *offset* and ends at *endTime*. When *looped* is TRUE, the first iteration of the loop begins at *offset* and ends at *endTime*. All subsequent repetitions begin at *startTime* and end at *endTime*.

### Example

The following Lingo adds the motion named Fall to the end of the bonesPlayer playlist of the model named Walker. When all motions before Fall in the playlist have been executed, Fall will play one time from beginning to end.

```
sprite(1).member.model("Walker").bonesPlayer.queue\  
    ("Fall", 0, 0, -1, 1, 0)
```

The following Lingo adds the motion named Kick to the end of the bonesPlayer playlist of the model named Walker. When all motions before Kick in the playlist have been executed, a section of Kick will play in a continuous loop. The first iteration of the loop will begin 2000 milliseconds from the motion's beginning. All subsequent iterations of the loop will begin 1000 milliseconds from Kick's beginning and will end 5000 milliseconds from Kick's beginning. The rate of playback will be three times the *playRate* property of the model's bonesPlayer modifier.

```
sprite(1).member.model("Walker").bonesPlayer.queue("Kick", 1, \  
    1000, 5000, 3, 2000)
```

### See also

play() (3D), playNext() (3D), playRate

## quickTimeVersion()

### Syntax

```
quickTimeVersion()
```

### Description

Function; returns a floating-point value that identifies the current installed version of QuickTime and replaces the current `QuickTimePresent` function.

In Windows, if multiple versions of QuickTime 3.0 or later are installed, `quickTimeVersion()` returns the latest version number. If a version before QuickTime 3.0 is installed, `quickTimeVersion()` returns version number 2.1.2 regardless of the version installed.

### Example

This statement uses `quickTimeVersion()` to display in the Message window the version of QuickTime that is currently installed:

```
put quickTimeVersion()
```

## quit

### Syntax

```
quit
```

### Description

Command; exits from Director or a projector to the Windows desktop or Macintosh Finder.

### Example

This statement tells the computer to exit to the Windows desktop or Macintosh Finder when the user presses Control+Q in Windows or Command+Q on the Macintosh:

```
if the key = "q" and the commandDown then quit
```

### See also

```
restart, shutdown
```

## QUOTE

### Syntax

```
QUOTE
```

### Description

Constant; represents the quotation mark character and refers to the literal quotation mark character in a string, because the quotation mark character itself is used by Lingo scripts to delimit strings.

### Example

This statement inserts quotation mark characters in a string:

```
put "Can you spell" && QUOTE & "Macromedia" & QUOTE & "?"
```

The result is a set of quotation marks around the word *Macromedia*:

```
Can you spell "Macromedia"?
```

## radius

### Syntax

```
modelResourceObjectReference.radius  
member(whichCastmember).modelResource(whichModelResource).radius
```

### Description

3D model property; when used with model resource of type `#sphere` or `#cylinder`, allows you to get or set the radius of the model.

The `radius` property determines the sweep radius used to generate the model resource. This property's value must always be set to greater than 0.0, and has a default value of 25.0.

### Example

This example shows that the radius of the model resource `Sphere01` is 24.0:

```
put member("3D World").modelResource("Sphere01").radius  
-- 24.0
```

## ramNeeded()

### Syntax

```
ramNeeded (firstFrame, lastFrame)
```

### Description

Function; determines the memory needed, in bytes, to display a range of frames. For example, you can test the size of frames containing 32-bit artwork: if `ramNeeded()` is larger than `freeBytes()`, then go to frames containing 8-bit artwork and divide by 1024 to convert bytes to kilobytes (K).

### Examples

This statement sets the variable `frameSize` to the number of bytes needed to display frames 100 to 125 of the movie:

```
put ramNeeded (100, 125) into frameSize
```

This statement determines whether the memory needed to display frames 100 to 125 is more than the available memory, and, if it is, branches to the section using cast members that have lower color depth:

```
if ramNeeded (100, 125) > the freeBytes then play frame "8-bit"
```

### See also

`freeBytes()`, `size`

# random()

## Syntax

`random(integerExpression)`

## Description

Function; returns a random integer in the range 1 to the value specified by *integerExpression*. This function can be used to vary values in a movie, such as to vary the path through a game, assign random numbers, or change the color or position of sprites.

To start a set of possible random numbers with a number other than 1, subtract the appropriate amount from the `random()` function. For example, the expression `random(n + 1) - 1` uses a range from 0 to the number *n*.

## Examples

This statement assigns random values to the variable `diceRoll`:

```
set diceRoll = random(6) + random(6)
```

This statement randomly changes the foreground color of sprite 10:

```
sprite(10).forecolor = random(256) - 1
```

This handler randomly chooses which of two movie segments to play:

```
on SelectScene
  if random(2) = 2 then
    play frame "11a"
  else
    play frame "11-b"
  end if
end
```

The following statements produce results in a specific range.

This statement produces a random multiple of 5 in the range 5 to 100:

```
theScore = 5 * random(20)
```

This statement produces a random multiple of 5 in the range 0 to 100:

```
theScore = 5 * (random(21) - 1)
```

This statement generates integers between -10 and +10:

```
dirH = random(21) - 11
```

This statement produces a random two-point decimal value:

```
the floatPrecision = 2
theCents = random(100)/100.0 - .01
```



## randomSeed

### Syntax

the randomSeed

### Description

System property; specifies the seed value used for generating random numbers accessed through the `random()` function.

Using the same seed produces the same sequence of random numbers. This property can be useful for debugging during development. Using the `ticks` property is an easy way to produce a unique random seed since the `ticks` value is highly unlikely to be duplicated on subsequent uses.

This property can be tested and set.

### Example

This statement displays the random seed number in the Message window:

```
put the randomSeed
```

### See also

`random()`, `ticks`

## randomVector

### Syntax

`randomVector()`

### Description

3D command; returns a unit vector describing a randomly chosen point on the surface of a unit sphere. This method differs from `vector(random(10)/10.0, random(10)/10.0, random(10)/10.0)` in that the resulting vector is guaranteed to be a unit vector.

### Examples

These statements create and display two randomly defined unit vectors in the Message window:

```
vec = randomVector()
put vec
-- vector(-0.1155, 0.9833, -0.1408)
vec2 = randomVector()
put vec2
-- vector(0.0042, 0.8767, 0.4810)
```

### See also

`getNormalized`, `generateNormals()`, `normalize`

## rawNew()

### Syntax

```
parentScript.rawNew()  
rawNew(parentScript)
```

### Description

Function; creates a child object from a parent script without calling its `on new` handler. This allows a movie to create child objects without initializing the properties of those child objects. This is particularly useful when you want to create large numbers of child objects for later use. To initialize the properties of one of these raw child objects, call its `on new` handler.

### Examples

This statement creates a child object called `RedCar` from the parent script `CarParentScript` without initializing its properties:

```
RedCar = script("CarParentScript").rawNew()
```

This statement initializes the properties of the child object `RedCar`:

```
RedCar.new()
```

### See also

`new()`, `script`

## realPlayerNativeAudio()

### Syntax

```
realPlayerNativeAudio()
```

### Description

`RealMedia` function; allows you to get or set the global flag that determines whether the audio portion of the `RealMedia` cast member is processed by `RealPlayer` (`TRUE`) or by `Director` (`FALSE`). This function returns the previous value of the flag.

To be effective, this flag must be set before `RealPlayer` is first loaded (when the first `RealMedia` cast member is encountered in the Score or with the first Lingo reference to a `RealMedia` cast member); any changes to this flag after `RealPlayer` is loaded are ignored. This flag should be executed in a `prepareMovie` event handler in a movie script. This flag is set for the entire session (from the time the Shockwave player is launched until it is closed and relaunched), not just for the duration of the current movie.

By default, this flag is set to `FALSE` and audio is processed by `Director`, which allows you to set the `soundChannel` (`RealMedia`) property and use the standard Lingo sound methods and properties to manipulate the audio stream of a `RealMedia` sprite, including mixing `RealAudio` with other `Director` audio. If this flag is set to `TRUE`, Lingo control of the sound channel is not processed, and the sound is handled by `RealPlayer`. For more information about working with `RealAudio`, see “Using `RealMedia` content in `Director`” in *Using Director*. For more information about working with `Director` audio, see the sound methods and properties in the main *Lingo Dictionary*.

### Examples

The following code shows that the `realPlayerNativeAudio()` function is set to `FALSE`, which means that audio in the `RealMedia` cast member will be processed by `Director`:

```
put realPlayerNativeAudio()  
-- 0
```

The following code sets the `realPlayerNativeAudio()` function to `TRUE`, which means that audio in the RealMedia stream will be processed by RealPlayer and all Lingo control of the sound channel will be ignored:

```
realPlayerNativeAudio(TRUE)
```

**See also**

`soundChannel (RealMedia)`, `audio (RealMedia)`

## **realPlayerPromptToInstall()**

**Syntax**

```
realPlayerPromptToInstall()
```

**Description**

RealMedia function; allows you to get or set a global flag that determines whether automatic detection and alert for RealPlayer 8 is enabled (`TRUE`) or not (`FALSE`).

By default, this function is set to `TRUE`, which means that if users do not have RealPlayer 8 and attempt to load a movie containing RealMedia, they are automatically asked if they want to go to the RealNetworks website and install RealPlayer. You can set this flag to `FALSE` if you want to create your own detection and alert system using the `realPlayerVersion()` function and custom code. If this flag is set to `FALSE` and an alternate RealPlayer 8 detection and alert system is not in place, users without RealPlayer will be able to load movies containing RealMedia cast members, but the RealMedia sprites will not appear.

This function detects the build number of the RealPlayer installed on the user's system to determine whether RealPlayer 8 is installed. On Windows systems, build numbers 6.0.8.132 or later indicate that RealPlayer 8 is installed. On Macintosh systems, RealPlayer Core component build numbers 6.0.7.1001 or later indicate that RealPlayer 8 is installed.

This flag should be executed in a `prepareMovie` event handler in a movie script.

This function returns the previous value of the flag.

**Examples**

The following code shows that the `realPlayerPromptToInstall()` function is set to `TRUE`, which means users who do not have RealPlayer will be prompted to install it:

```
put realPlayerPromptToInstall()  
-- 1
```

The following code sets the `realPlayerPromptToInstall()` function to `FALSE`, which means that users will not be prompted to install RealPlayer unless you have created a detection and alert system:

```
realPlayerPromptToInstall(FALSE)
```

# realPlayerVersion()

## Syntax

`realPlayerVersion()`

## Description

RealMedia function; returns a string identifying the build number of the RealPlayer software installed on the user's system, or an empty string if RealPlayer is not installed. Users must have RealPlayer 8 or later in order to view Director movies containing RealMedia content. On Windows systems, build numbers 6.0.8.132 or later indicate that RealPlayer 8 is installed. On Macintosh systems, RealPlayer Core component build numbers 6.0.7.1001 or later indicate that RealPlayer 8 is installed.

The purpose of this function is to allow you to create your own RealPlayer detection and alert system, if you do not want to use the one provided by the `realPlayerPromptToInstall()` function.

If you choose to create your own detection and alert system using the `realPlayerVersion()` function, you must do the following:

- Call `realPlayerPromptToInstall(FALSE)` (by default, this function is set to `TRUE`) before any RealMedia cast members are referenced in Lingo or appear in the Score. This function should be set in a `prepareMovie` event handler in a movie script.
- Use the `xtraList` system property to verify that the Xtra for RealMedia (RealMedia Asset.x32) is listed in the Movie Xtras dialog box. The `realPlayerVersion()` function will not work if the Xtra for RealMedia is not present.

The build number returned by this function is the same as the build number you can display in RealPlayer.

### To view the RealPlayer build number in Windows:

- 1 Launch RealPlayer.
- 2 Choose About RealPlayer from the Help menu.

In the window that appears, the build number appears at the top of the screen in the second line.

### To view the RealPlayer build number on the Macintosh:

- 1 Launch RealPlayer.
- 2 Choose About RealPlayer from the Apple menu.

The About RealPlayer dialog box appears. Ignore the build number listed in the second line at the top of the screen; it is incorrect.

- 3 Click the Version Info button.

The RealPlayer Version Information dialog box appears.

- 4 Select RealPlayer Core in the list of installed components.

The build number shown for RealPlayer Core component (for example, 6.0.8.1649) is the same as the build number returned by `realPlayerVersion()`.

### Example

The following code shows that build number of the RealPlayer installed on the system is 6.0.9.357:

```
put realPlayerVersion()  
-- "6.0.9.357"
```

## recordFont

### Syntax

```
recordFont(whichCastMember, font {[face]} {[bitmapSizes]} [characterSubset] [userFontName])
```

### Description

Command; embeds a TrueType or Type 1 font as a cast member. Once embedded, these fonts are available to the author just like other fonts installed in the system.

You must create an empty font cast member with the `new()` command before using `recordFont`.

- *font*—Name of original font to be recorded.
- *face*—List of symbols indicating the face of the original font; possible values are `#plain`, `#bold`, `#italic`. If you do not provide a value for this argument, `#plain` is used.
- *bitmapSizes*—List of integers specifying the sizes for which bitmaps are to be recorded. This argument can be empty. If you omit this argument, no bitmaps are generated. These bitmaps typically look better at smaller point sizes (below 14 points) but take up more memory.
- *characterSubset*—String of characters to be encoded. Only the specified characters will be available in the font. If this argument is, all characters are encoded. If only certain characters are encoded but an unencoded character is used, that character is displayed as an empty box.
- *userFontName*— A string to use as the name of the newly recorded font cast member.

The command creates a Shock Font in *whichCastMember* using the font named in the *font* argument. The value returned from the command reports whether the operation was successful. Zero indicates success.

### Examples

This statement creates a simple Shock Font using only the two arguments for the cast member and the font to record:

```
myNewFontMember = new(#font)  
recordFont(myNewFontMember, "Lunar Lander")
```

This statement specifies the bitmap sizes to be generated and the characters for which the font data should be created:

```
myNewFontMember = new(#font)  
recordFont(myNewFontMember, "lunar lander", [], [14, 18, 45], "Lunar Lander Game High \  
Score First Last Name")
```

**Note:** Since `recordFont` resynthesizes the font data rather than using it directly, there are no legal restrictions on Shock Font distribution.

### See also

`new()`

## rect (camera)

### Syntax

```
sprite(whichSprite).camera(whichCamera).rect
```

### Description

3D camera property; allows you to get or set the rectangle that controls the size and position of the camera. This rectangle is analogous to the rectangle you see through the eyepiece of a real camera.

The default value for the rect property for all cameras rect(0,0,1,1) which makes them invisible until you change the setting. However, when `sprite.camera(1)` is rendered, its rect is reset to `rect(0,0,sprite(whichSprite).width,sprite(whichSprite).height)` so that the camera fills the screen. All camera rect coordinates are given relative to the top left corner of the sprite.

Note that if *whichCamera* is greater than 1, the rect is not scaled when the sprite is scaled, so it will be necessary to manage that in Lingo if desired.

When *whichCamera* is greater than 1, the `rect.top` and `rect.left` properties must be greater than or equal to the `rect.top` and `rect.left` settings for `sprite.camera(1)`.

### Example

This statement sets the rect of the default camera of sprite 5 to rect(0, 0, 200, 550):

```
sprite(5).camera.rect = rect(0, 0, 200, 550)
```

### See also

`cameraPosition`, `cameraRotation`

## rect()

### Syntax

```
rect(left, top, right, bottom)  
rect(point1, point2)
```

### Description

Function and data type; defines a rectangle.

The `rect(left, top, right, bottom)` format defines a rectangle whose sides are specified by *left*, *top*, *right*, and *bottom*. The *left* and *right* values specify numbers of pixels from the left edge of the Stage. The *top* and *bottom* values specify numbers of pixels from the top of the Stage.

The `rect(point1, point2)` format defines a rectangle that encloses the points specified by *point1* and *point2*.

You can refer to rectangle components by list syntax or property syntax. For example, the following two phrases are equivalent:

```
targetWidth = targetRect.right - targetRect.left  
targetWidth = targetRect[3] - targetRect[1]
```

You can perform arithmetic operations on rectangles. If you add a single value to a rectangle, Lingo adds it to each element in the rectangle.

To see an example of `rect()` used in a completed movie, see the Imaging movie in the Learning/Lingo Examples folder inside the Director application folder.

### Examples

This statement sets the variable `newArea` to a rectangle whose left side is at 100, top is at 150, right side is at 300, and bottom is at 400 pixels:

```
set newArea = rect(100, 150, 300, 400)
```

The following statement sets the variable `newArea` to the rectangle defined by the points `firstPoint` and `secondPoint`. The coordinates of `firstPoint` are (100, 150); the coordinates of `secondPoint` are (300, 400). Note that this statement creates the same rectangle as the one created in the previous example:

```
put rect(firstPoint, secondPoint)
```

These statements add and subtract values for rectangles:

```
put rect(0,0,100,100) + rect(30, 55, 120, 95)
-- rect(30, 55, 220, 195)
put rect(0,0,100,100) - rect(30, 55, 120, 95)
-- rect(-30, -55, -20, 5)
```

This statement adds 80 to each coordinate in a rectangle:

```
put rect(60, 40, 120, 200) + 80
-- rect(140, 120, 200, 280)
```

This statement divides each coordinate in a rectangle by 3:

```
put rect(60, 40, 120, 200) / 3
-- rect(20, 13, 40, 66)
```

### See also

`point()`, `quad`

## rect (image)

### Syntax

*imageObject*.rect

### Description

Read-only property; returns a rectangle describing the size of the given image object. The coordinates are given relative to the top left corner of the image. The left and top values of the rectangle are therefore always 0, and the right and bottom values are the width and height of the cast member.

### Examples

This statement returns the rectangle of the 300 x 400 pixel member `Sunrise` in the message window:

```
put member("Sunrise").image.rect
-- rect(0, 0, 300, 400)
```

This Lingo looks at the first 50 cast members and displays the rectangle and name of each cast member that is a bitmap:

```
on showAllRects
  repeat with x = 1 to 50
    if member(x).type = #bitmap then
      put member(x).image.rect && "-" && member(x).name
    end if
  end repeat
end
```

**See also**

height,width

## rect (member)

**Syntax**

```
member(whichCastMember).rect  
the rect of member whichCastMember
```

**Description**

Cast member property; specifies the left, top, right, and bottom coordinates, returned as a rectangle, for the rectangle of any graphic cast member, such as a bitmap, shape, movie, or digital video.

For a bitmap, the `rect` member property is measured from the upper left corner of the bitmap, instead of from the upper left corner of the easel in the Paint window.

For an Xtra cast member, the `rect` member property is a rectangle whose upper left corner is at (0,0).

The Director player for Java can't set the `rect` member property.

This property can be tested. It can be set for field cast members only.

**Examples**

This statement displays the coordinates of bitmap cast member 20:

```
put member(20).rect
```

This statement sets the coordinates of bitmap cast member Banner:

```
member("Banner").rect = rect(100, 150, 300, 400)
```

**See also**

`rect()`, `rect (sprite)`

## rect (sprite)

**Syntax**

```
sprite whichSprite.rect  
the rect of sprite whichSprite
```

**Description**

Sprite property; specifies the left, top, right, and bottom coordinates, as a rectangle, for the rectangle of any graphic sprite such as a bitmap, shape, movie, or digital video.

This property can be tested and set.

**Example**

This statement displays the coordinates of bitmap sprite 20:

```
put sprite(20).rect
```



## rect (window)

### Syntax

*window* *whichWindow*.rect  
the rect of window *whichWindow*

### Description

Window property; specifies the left, top, right, and bottom coordinates, as a rectangle, of the window specified by *whichWindow*.

If the size of the rectangle specified is less than that of the Stage where the movie was created, the movie is cropped in the window, not resized.

To pan or scale the movie playing in the window, set the `drawRect` or `sourceRect` property of the window.

This property can be tested and set.

### Example

This statement displays the coordinates of the window named `Control_panel`:

```
put window("Control_panel").rect
```

### See also

`drawRect`, `sourceRect`

## ref

### Syntax

*chunkExpression*.ref

### Description

Text chunk expression property; this provides a convenient way to refer to a chunk expression within a text cast member.

### Examples

Without references, you would need statements like these:

```
member(whichTextMember).line[whichLine].word[firstWord..lastWord].font = #Palatino"
member(whichTextMember).line[whichLine].word[firstWord..lastWord].fontSize = 36
member(whichTextMember).line[whichLine].word[firstWord..lastWord].fontStyle = #Bold]
```

But with a `ref` property, you can refer to the same chunk as follows:

```
myRef = member(whichTextMember).line[whichLine].word[firstWord..lastWord].ref
```

The variable `myRef` is now shorthand for the entire chunk expression. This allows something like the following:

```
put myRef.font
-- "Palatino"
```

Or you can set a property of the chunk as follows:

```
myRef.fontSize = 18
myRef.fontStyle = [#italic]
```

You can get access to the string referred to by the reference using the text property of the reference:

```
put myRef.text
```

This would result in the actual string data, not information about the string.

## reflectionMap

### Syntax

```
member(whichCastmember).shader(whichShader).reflectionMap
```

### Description

3D shader property; allows you to get and set the texture used to create reflections on the surface of a model. This texture is applied to the third texture layer of the shader. This property is ignored if the `toon` modifier is applied to the model resource.

This helper property provides a simple interface for setting up a common use of reflection mapping. The same effect can be achieved by setting the following properties:

```
shader.textureModelList[3] = #reflection
shader.blendFunctionList[3] = #blend
shader.blendSourceList[3] = #constant
shader.blendConstantList[3] = 50.0
```

When tested, this property returns the texture associated with the model's third texture layer. The default is `void`.

### Example

This statement causes the model named `GlassSphere` to appear to reflect the texture named `Portrait` off of its surface:

```
member("3DPlanet").model("GlassSphere").shader.reflectionMap = \
    member("3DPlanet").texture("Portrait")
```

### See also

```
textureModelList, blendFunctionList, blendConstantList
```

## reflectivity

### Syntax

```
member(whichCastmember).reflectivity
```

### Description

3D shader property; allows you to get or set the shininess of the referenced member's default shader. The value is a floating point value representing the percentage of light to be reflected off the surface of a model using the default shader, from 0.0 to 100.00. The default value is 0.0.

### Example

This statement sets the shininess of the default shader in the cast member named `Scene` to 50%:

```
member("Scene").reflectivity = 50
```

## region

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
    emitter.region
modelResourceObjectReference.emitter.region
```

### Description

3D emitter property; when used with a model resource whose type is #particle, allows you to both get and set the `region` property of the resource's particle emitter.

The `region` property defines the location from which particles are emitted. If its value is a single vector, then that vector is used to define a point in the 3D world from which particles will be emitted.

If its value is a list of two vectors, then those vectors are used to define the end points of a line segment from which particles will be emitted.

If its value is a list of four vectors, then those vectors are used to define the vertices of a quadrilateral from which the particles will be emitted.

The default value for this property is [vector(0,0,0)].

### Example

In this example, `ThermoSystem` is a model resource of the type #particle. This statement specifies the four corners of a rectangle from which the particles of `ThermoSystem` originate.

```
member("Fires").modelResource("ThermoSystem").emitter.region = \
    [vector(20,90,100), vector(30,90,100), vector(30,100,100), \
    vector(20,100,100)]
```

### See also

`emitter`

## registerForEvent()

### Syntax

```
member(whichCastmember).registerForEvent(eventName, \
    handlerName, scriptObject {, begin, period, repetitions})
```

### Description

3D command; declares the specified handler as the handler to be called when the specified event occurs within the specified cast member.

The following parameter descriptions apply to both the `registerForEvent()` and the `registerScript()` commands.

The *handlerName* parameter is the name of the handler that will be called; this handler is found in the script object indicated by *scriptObject*.

If 0 is specified for *scriptObject*, then the first event handler with the given name found in a movie script is called.

The *eventName* parameter can be any of the following predefined Lingo events, or any custom event that you define:

- *#collideAny* is a collision event.
- *#collideWith* is a collision event involving this specific model. The *setCollisionCallback()* command is a shortcut for using the *registerScript()* command for the *#collideWith* event.
- *#animationStarted* and *#animationEnded* are notification events that occur when a bones or keyframe animation starts or stops playing. The handler will receive three arguments: *eventName*, *motion*, and *time*. The *eventName* argument is either *#animationStarted* or *#animationEnded*. The *motion* argument is the name of the motion that has started or stopped playing, and *time* is the current time of the motion.

For looping animations, the *#animationStarted* event is issued only for the first loop, not for subsequent loops. During a blend of two animations, this event will be sent when the blending begins.

When a series of animations is queued for the model and the animation's *autoBlend* property is set to *TRUE*, the *#animationEnded* event may occur before the apparent end of a given motion. This is because the *autoBlend* property may make the motion appear to continue even though the animation has completed as defined.

- *#timeMS* is a time event. The first *#timeMS* event occurs when the number of milliseconds specified in the *begin* parameter have elapsed after *registerForEvent* is called. The *period* parameter determines the number of milliseconds between *#timeMS* events when the value of *repetitions* is greater than 0. If *repetitions* is 0, the *#timeMS* event occurs indefinitely.

The handler you specify is sent the following arguments:

*type* is always 0.

*delta* is the elapsed time in milliseconds since the last *#timeMS* event.

*time* is the number of milliseconds since the first *#timeMS* event occurred. For example, if there are three iterations with a period of 500 ms, the first iteration's time will be 0, the second iteration will be 500, and the third will be 1000.

*duration* is the total number of milliseconds that will elapse between the *registerForEvent* call and the last *#timeMS* event. For example, if there are five iterations with a period of 500 ms, the duration is 2500 ms. For tasks with unlimited iterations, the duration is 0.

*systemTime* is the absolute time in milliseconds since the Director movie started.

**Note:** You can associate the registration of a script with a particular node rather than with a cast member by using the *registerScript()* command.

### Examples

This statement registers the *promptUser* event handler found in a movie script to be called twice at an interval of 5 seconds:

```
member("Scene").registerForEvent(#timeMS, #promptUser, 0, \
    5000, 5000, 2)
```

This statement registers the *promptUser* event handler found in a movie script to be called each time a collision occurs within the cast member named *Scene*:

```
member("Scene").registerForEvent(#collideAny, #promptUser, 0)
```

This statement declares the `on promptUser` handler in the same script that contains the `registerForEvent` command to be called when any object collides with the model named Pluto in the cast member named Scene:

```
member("Scene").registerForEvent(#collideWith, #promptUser, me, \
    member("Scene").model("Pluto"))
```

**See also**

`setCollisionCallback()`, `registerScript()`, `play()` (3D), `playNext()` (3D), `autoblend`, `blendTime`, `sendEvent`, `unregisterAllEvents`

## registerScript()

**Syntax**

```
member(whichCastmember).model(whichModel).registerScript(eventName, \
    handlerName, scriptObject {, begin, period, repetitions})
member(whichCastmember).camera(whichCamera).registerScript(eventName, \
    handlerName, scriptObject {, begin, period, repetitions})
member(whichCastmember).light(whichLight).registerScript(eventName, \
    handlerName, scriptObject {, begin, period, repetitions})
member(whichCastmember).group(whichGroup).registerScript(eventName, \
    handlerName, scriptObject {, begin, period, repetitions})
```

**Description**

3D command; registers the specified handler to be called when the specified event occurs for the referenced node.

The following parameter descriptions apply to both the `registerForEvent()` and the `registerScript()` commands.

The *handlerName* parameter is the name of the handler that will be called; this handler is to be found in the script object indicated by *scriptObject*.

If 0 is specified for *scriptObject*, then the first event handler with the given name found in a movie script is called.

The *eventName* parameter can be any of the following predefined Lingo events, or any custom event that you define:

- `#collideAny` is a collision event generated when any two bodies in the system collide with each other, and both bodies have the `#collision` modifier attached.
- `#collideWith` is a collision event involving this specific model. The `setCollisionCallback()` command is a shortcut for using the `registerScript()` command for the `#collideWith` event.

- `#animationStarted` and `#animationEnded` are notification events that occur when a bones or keyframe animation starts or stops playing. The handler will receive three arguments: *eventName*, *motion*, and *time*. The *eventName* argument is either `#animationStarted` or `#animationEnded`. The *motion* argument is the name of the motion that has started or stopped playing, and *time* argument is the current time of the motion.

For looping animations, the `#animationStarted` event is issued only for the first loop, not for subsequent loops. During a blend of two animations, this event will be sent as the blend begins.

When a series of animations is queued for the model and the animation's `autoBlend` property is set to `TRUE`, the `#animationEnded` event may occur before the apparent end of a given motion. This is because the `autoBlend` property may make the motion appear to continue even though the animation has completed as defined.

- `#timeMS` is a time event. The first `#timeMS` event occurs when the number of milliseconds specified in the *begin* parameter have elapsed after `registerForEvent` is called. The *period* parameter determines the number of milliseconds between `#timeMS` events when the value of *repetitions* is greater than 0. If *repetitions* is 0, the `#timeMS` event occurs indefinitely.

The handler you specify is sent the following arguments:

*type* is always 0.

*delta* is the elapsed time in milliseconds since the last `#timeMS` event.

*time* is the number of milliseconds since the first `#timeMS` event. For example, if there are three iterations with a period of 500 ms, the first iteration's time will be 0, the second iteration will be 500, and the third will be 1000.

*duration* is the total number of milliseconds that will elapse between the `registerForEvent` call and the last `#timeMS` event. For example, if there are five iterations with a period of 500 ms, the duration is 2500 ms. For tasks with unlimited iterations, the duration is 0.

*systemTime* is the absolute time in milliseconds since the Director movie started.

### Examples

This statement registers the `messageReceived` event handler found in a movie script to be called when the model named `Player` receives the custom user defined event named `#message`:

```
member("Scene").model("Player").registerScript(#message, \
    #messageReceived, 0)
```

This statement registers the `collisionResponder` event handler found in the same script as the `registerScript` command to be called each time a collision occurs between the model named `Player` and any other model using the `#collision` modifier:

```
member("Scene").model("Player").registerScript(#collideWith, \
    #collisionResponder, me)
```

### See also

`registerForEvent()`, `sendEvent`, `setCollisionCallback()`, `collisionData`

# regPoint

## Syntax

`member(whichCastMember).regPoint`  
the `regPoint` of member *whichCastMember*

## Description

Cast member property; specifies the registration point of a cast member. The registration point is listed as the horizontal and vertical coordinates of a point in the form `point(horizontal, vertical)`. Nonvisual members such as sounds do not have a useful `regPoint` property.

You can use the `regPoint` property to animate individual graphics in a film loop, changing the film loop's position in relation to other objects on the Stage.

You can also use `regPoint` to adjust the position of a mask being used on a sprite.

When a Flash movie cast member is first inserted into the cast, its registration point is its center and its `centerRegPoint` property is set to `TRUE`. If you subsequently use the `regPoint` property to reposition the registration point, the `centerRegPoint` property is automatically set to `FALSE`.

This property can be tested and set.

## Examples

This statement displays the registration point of the bitmap cast member `Desk` in the Message window:

```
put member("Desk").regPoint
```

This statement changes the registration point of the bitmap cast member `Desk` to the values in the list:

```
member("Desk").regPoint = point(300, 400)
```

## See also

`centerRegPoint`, `mask`

# regPoint (3D)

## Syntax

`sprite(whichSprite).camera.backdrop[backdropIndex].regPoint`  
`member(whichCastmember).camera(whichCamera).backdrop[backdropIndex].regPoint`

## Description

3D backdrop and overlay property; allows you to get or set the registration point of the backdrop or overlay. The registration point represents the *x*, *y*, and *z* coordinates of the center of the backdrop or overlay in 3D space. The default value for this property is `point(0,0)`.

## Example

The following statement changes the registration point of the first backdrop of the camera of sprite 13. The backdrop's registration point will be the point (50, 0), measured from the upper left corner of the backdrop.

```
sprite(13).camera.backdrop[1].regPoint = point(50, 0)
```

## See also

`loc` (backdrop and overlay)

## regPointVertex

### Syntax

*vectorMember*.regPointVertex  
the regPointVertex of *vectorMember*

### Description

Cast member property; indicates whether a vertex of *vectorCastMember* is used as the registration point for that cast member. If the value is zero, the registration point is determined normally, using the centerRegPoint and regPoint properties. If the value is nonzero, it indicates the position in the vertexList of the vertex being used as the registration point. The centerRegPoint is set to FALSE and the regPoint is set to the location of that vertex.

### Example

This statement makes the registration point for the vector shape cast member Squiggle correspond to the location of the third vertex:

```
member("squiggle").regPointVertex=3  
centerRegPoint, regPoint
```

## relative

### See

@ (pathname)

## removeBackdrop

### Syntax

member(*whichCastmember*).camera(*whichCamera*).removeBackdrop(*index*)

### Description

3D command; removes the backdrop found in the position specified by *index* from the camera's list of backdrops to display.

### Example

The following statement removes the third backdrop from the list of backdrops for camera 1 within the member named Scene. The backdrop will disappear from the stage if there are any sprites currently using this camera.

```
member("Scene").camera[1].removeBackdrop(3)
```

## removeFromWorld

### Syntax

```
member(whichCastmember).model(whichModel).removeFromWorld()  
member(whichCastmember).light(whichLight).removeFromWorld()  
member(whichCastmember).camera(whichCamera).removeFromWorld()  
member(whichCastmember).group(whichGroup).removeFromWorld()
```

### Description

3D command; for models, lights, cameras or groups whose parent hierarchy terminates in the world object, this command sets their parent to void and removes them from the world.



For objects whose parent hierarchy does not terminate in the world, this command does nothing.

#### Example

This command removes the model named `gbCyl` from the 3D world of the cast member named `Scene`:

```
member("Scene").model("gbCyl").removeFromWorld()
```

## removeLast()

```
member(whichCastmember).model(whichModel).bonesPlayer.removeLast()  
member(whichCastmember).model(whichModel).keyframePlayer.\  
    removeLast()
```

#### Description

3D `keyframePlayer` and `bonesPlayer` modifier command; removes the last motion from the modifier's playlist.

#### Example

This statement removes the last motion from the playlist of the `bonesPlayer` modifier for the model named `Walker`:

```
member("MyWorld").model("Walker").bonesPlayer.removeLast()
```

## removeModifier

#### Syntax

```
member(whichCastmember).model(whichModel).removeModifier.\  
    (#whichModifier)
```

#### Description

3D command; removes the modifier identified by *#whichModifier* from the specified model.

This command returns `TRUE` if it completes successfully, and `FALSE` if *#whichModifier* is not a valid modifier, or if the modifier was not attached to the model.

#### Example

This statement removes the *#toon* modifier from the model named `Box`:

```
member("shapes").model("Box").removeModifier(#toon)
```

#### See also

`addModifier`, `modifier`, `modifier[]`, `modifiers`

## removeOverlay

### Syntax

```
member(whichCastmember).camera(whichCamera).removeOverlay(index)
```

### Description

3D command; removes the overlay found in the position specified by *index* from the camera's list of overlays to display.

### Example

The following statement removes the third overlay from the list of overlays for the camera being used by sprite 5. The overlay disappears from the Stage.

```
sprite(5).camera.removeOverlay(1)
```

### See also

overlay

## renderer

### Syntax

```
getRendererServices().renderer
```

### Description

3D property; allows you to get or set the current renderer in use by a movie. The range of values for this property is determined by the list of available renderers returned by the Renderer Services object's `rendererDeviceList` property.

Shockwave users have the option of specifying the renderer of their choice using the 3D Renderer context menu in Shockwave. If the user selects the “Obey content settings” option, the renderer specified by the `renderer` or `preferred3DRenderer` properties is used to draw the movie (if available on the users system), otherwise the renderer selected by the user is used.

The default value for this property is determined by the `preferred3DRenderer` property.

This property returns the same value as returned by the movie property the `active3dRenderer`.

### Example

This statement shows that the renderer currently being used by the user's system is `#openGL`:

```
put getRendererServices().renderer  
-- #openGL
```

### See also

`getRendererServices()`, `preferred3DRenderer`, `rendererDeviceList`, `active3dRenderer`

## rendererDeviceList

### Syntax

```
getRendererServices().rendererDeviceList
```

### Description

3D renderer property; returns a list of symbols identifying the renderers that are available for use on the client machine. The contents of this list determine the range of values that can be specified for the `renderer` and `preferred3DRenderer` properties. This property can be tested but not set.

This property is a list that can contain the following possible values:

- `#openGL` specifies the openGL drivers for a hardware acceleration which work with both Macintosh and Windows platforms.
- `#directX7_0` specifies the DirectX 7 drivers for hardware acceleration which work with Windows platforms only.
- `#directX5_2` specifies the DirectX 5.2 drivers for hardware acceleration which work with Windows platforms only.
- `#software` specifies the Director built-in software renderer which works with both Macintosh and Windows platforms.

#### Example

This statement shows the renderers available on the current system:

```
put getRendererServices().rendererDeviceList
-- [#openGL, #software]
```

#### See also

`getRendererServices()`, `renderer`, `preferred3DRenderer`, `active3DRenderer`

## renderFormat

#### Syntax

```
member(whichCastmember).texture(whichTexture).renderFormat
member(whichCastmember).texture[index].renderFormat
member(whichCastmember).shader(whichShader).texture.renderFormat
member(whichCastmember).model(whichModel).shader.texture\
    .renderFormat
member(whichCastmember).model(whichModel).shader.textureList[
    index].renderFormat
member(whichCastmember).model(whichModel).shaderList[index].\
    texture(whichTexture).renderFormat
member(whichCastmember).model(whichModel).shaderList[index].\
    textureList[index].renderFormat
```

#### Description

3D property; allows you to get or set the `textureRenderFormat` for a specific texture by specifying one of the following values:

`#default` uses the value returned by `getRendererServices().textureRenderFormat`.

```
#rgba8888
#rgba8880
#rgba5650
#rgba5550
#rgba5551
#rgba4444
```

See `textureRenderFormat` for information on these values.

Setting this property for an individual texture overrides the global setting set using `textureRenderFormat`.

The `renderFormat` property determines the pixel format the renderer uses when rendering the specified texture. Each pixel format has a number of digits, with each digit indicating the color depth being used for red, green, blue, and alpha. The value you choose determines the accuracy of the color fidelity (including the precision of the optional alpha channel) and thus the amount of memory used on the video card. You can choose a value that improves color fidelity or a value that allows you to fit more textures into memory on the video card. You can fit roughly twice as many 16-bit textures as 32-bit textures in the same space.

#### Example

The following statement sets the `renderFormat` property of the texture `TexPic` to `#rgba4444`. The red, blue, green, and alpha components of the texture will each be drawn using 4 bits of information.

```
member("3d").texture("TexPic").renderFormat = #rgba4444
```

#### See also

`textureRenderFormat`, `getHardwareInfo()`

## renderStyle

#### Syntax

```
member(whichCastmember).shader(whichShader).renderStyle
```

#### Description

3D standard shader property; allows you to get or set the `renderStyle` for a shader, as determined by the geometry of the underlying model resource. This property has the following values:

**#fill** specifies that the shader is drawn to completely fill the surface area of the model resource.

**#wire** specifies that the shader is drawn only on the edges of the faces of the model resource.

**#point** specifies that the shader is drawn only on the vertices of the model resource.

All shaders have access to the `#standard` shader properties; in addition to these standard shader properties, shaders of the types `#engraver`, `#newsprint`, and `#painter` have properties unique to their type. For more information, see `newShader`.

#### Example

This statement causes the shader `WallMaterial` to be rendered only where it lies on top of a vertex of the underlying model resource:

```
member("CityScene").shader("WallMaterial").renderStyle = #point
```

## repeat while

#### Syntax

```
repeat while testCondition  
    statement(s)  
end repeat
```

#### Description

Keyword; repeatedly executes *statement(s)* so long as the condition specified by *testCondition* is TRUE. This structure can be used in Lingo that continues to read strings until the end of a file is reached, checks items until the end of a list is reached, or repeatedly performs an action until the user presses or releases the mouse button.

While in a repeat loop, Lingo ignores other events. To check the current key in a repeat loop, use the `keyPressed` property.

Only one handler can run at a time. If Lingo stays in a repeat loop for a long time, other events stack up waiting to be evaluated. Therefore, repeat loops are best used for short, fast operations or when users are idle.

If you need to process something for several seconds or more, evaluate the function in a loop with some type of counter or test to track progress.

The Director player for Java doesn't detect mouse movements, update properties that indicate the mouse's position, or update the status of mouse button presses when Lingo is in a repeat loop.

If the stop condition is never reached or there is no exit from the repeat loop, you can force Director to stop by using `Control+Alt+period` (Windows) or `Command+period` (Macintosh).

### Example

This handler starts the timer counting, resets the timer to 0, and then has the timer count up to 60 ticks:

```
on countTime
    startTimer
    repeat while the timer < 60
        -- waiting for time
    end repeat
end countTime
```

### See also

`exit`, `exit repeat`, `repeat with`, `keyPressed()`

## repeat with

### Syntax

```
repeat with counter = start to finish
    statement(s)
end repeat
```

### Description

**Keyword;** executes the Lingo specified by *statement(s)* the number of times specified by *counter*. The value of *counter* is the difference between the value specified by *start* and the value specified by *finish*. The counter is incremented by 1 each time Lingo cycles through the repeat loop.

The `repeat with` structure is useful for repeatedly applying the same effect to a series of sprites or for calculating a series of numbers to some exponent.

While in a repeat loop, Lingo ignores other events. To check the current key in a repeat loop, use the `keyPressed` property.

Only one handler can run at a time. If Lingo stays in a repeat loop for a long time, other events stack up waiting to be evaluated. Therefore, repeat loops are best used for short, fast operations or when users are idle.

If you need to process something for several seconds or more, evaluate the function in a loop with some type of counter or test to track progress.

If the stop condition is never reached or there is no exit from the repeat loop, you can force Director to stop by using `Control+Alt+period` (Windows) or `Command+period` (Macintosh).

The Director player for Java doesn't detect mouse movements, update properties that indicate the mouse's position, or update the status of mouse button presses when Lingo is in a repeat loop.

### Example

This handler turns sprites 1 through 30 into puppets:

```
on puppetize
    repeat with channel = 1 to 30
        puppetSprite channel, TRUE
    end repeat
end puppetize
```

### See also

exit, **exit repeat**, repeat while, repeat with...down to, repeat with...in list

## repeat with...down to

### Syntax

repeat with *variable* = *startValue* down to *endValue*

### Description

Keyword; counts down by increments of 1 from *startValue* to *endValue*.

Only one handler can run at a time. If Lingo stays in a repeat loop for a long time, other events stack up waiting to be evaluated. Therefore, repeat loops are best used for short, fast operations or when you know the user won't be doing other things.

While in a repeat loop, Lingo ignores other events. To check the current key in a repeat loop, use the `keyPressed` property.

If you need to process something for several seconds or more, evaluate the function in a loop with some type of counter or test to track progress.

If the stop condition is never reached or there is no exit from the repeat loop, you can force Director to stop by using Control+Alt+period (Windows) or Command+period (Macintosh).

The Director player for Java doesn't detect mouse movements, update properties that indicate the mouse's position, or update the status of mouse button presses when Lingo is in a repeat loop.

### Example

This handler contains a repeat loop that counts down from 20 to 15:

```
on Countdown
    repeat with i = 20 down to 15
        sprite(6).memberNum = 10 + i
        updateStage
    end repeat
end
```

## repeat with...in list

### Syntax

repeat with *variable* in *someList*

### Description

**Keyword;** assigns successive values from the specified list to the variable.

While in a repeat loop, Lingo ignores other events except keypresses. To check the current key in a repeat loop, use the `keyPressed` property.

Only one handler can run at a time. If Lingo stays in a repeat loop for a long time, other events stack up waiting to be evaluated. Therefore, repeat loops are best used for short, fast operations or when users are idle.

If you need to process something for several seconds or more, evaluate the function in a loop with some type of counter or test to track progress.

If the stop condition is never reached or there is no exit from the repeat loop, you can force Director to stop by using Control+Alt+period (Windows) or Command+period (Macintosh).

The Director player for Java doesn't detect mouse movements, update properties that indicate the mouse's position, or update the status of mouse button presses when Lingo is in a repeat loop.

### Example

This statement displays four values in the Message window:

```
repeat with i in [1, 2, 3, 4]
    put i
end repeat
```

## resetWorld

### Syntax

```
member(whichCastmember).resetWorld()
member(whichTextCastmember).resetWorld()
```

### Description

3D command; resets the member's properties of the referenced 3D cast member to the values stored when the member was first loaded into memory. The member's `state` property must be either 0 (unloaded), 4 (media loaded), or -1 (error) before this command can be used, otherwise a script error will occur.

This command differs from `revertToWorldDefaults` in that the values used are taken from the state of the member when it was first loaded into memory rather than from the state of the member when it was first created.

### Example

This statement resets the properties of the cast member named Scene to the values they had when the member was first loaded into memory:

```
member("Scene").resetWorld()
```

### See also

`revertToWorldDefaults`

## on resizeWindow

### Syntax

```
on resizeWindow  
    statement(s)  
end
```

### Description

System message and event handler; contains statements that run when a movie is running as a movie in a window (MIAW) and the user resizes the window by dragging the window's resize box or one of its edges.

An `on resizeWindow` event handler is a good place to put Lingo related to the window's dimensions, such as Lingo that positions sprites or crops digital video.

### Example

This handler moves sprite 3 to the coordinates stored in the variable `centerPlace` when the window that the movie is playing in is resized:

```
on resizeWindow centerPlace  
    sprite(3).loc = centerPlace  
end
```

### See also

`drawRect`, `sourceRect`

## resolution

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).resolution
```

### Description

3D property; allows you to get or set the resolution property of a model resource whose type is either `#sphere` or `#cylinder`.

Resolution controls the number of polygons used to generate the geometry of the model resource. A larger value generates more polygons and thus results in a smoother surface. The default value of this property is 20.

### Example

This statement sets the resolution of the model resource named `sphere01` to 10.0:

```
member("3D World").modelResource("sphere01").resolution = 10.0
```



## resolve

### Syntax

```
member(whichCastmember).model(whichModel).collision.resolve
```

### Description

3D collision property; allows you to get or set whether collisions are resolved when two models collide. If this property is set to `TRUE` for both models involved in a collision, both models come to a stop at the point of collision. If only one of the models has the `resolve` property set to `TRUE`, that model comes to a stop, and the model with the property not set, or set to `FALSE`, continues to move. The default value for this property is `TRUE`.

### Example

The following statement sets the `resolve` property of the collision modifier applied to the model named `Box` to `TRUE`. When the model named `Box` collides with another model that has the `#collision` modifier attached, it will stop moving.

```
member("3d world").model("Box").collision.resolve = TRUE
```

### See also

`collisionData`, `collisionNormal`, `modelA`, `modelB`, `pointOfContact`

## resolveA

### Syntax

```
collisionData.resolveA(bResolve)
```

### Description

3D collision method; overrides the collision behavior set by the `collision.resolve` property for `collisionData.modelA`. If *bResolve* is `TRUE`, then the collision for the `modelA` is resolved, if *bResolve* is `FALSE` the collision for `modelA` is not resolved. Call this function only if you wish to override the behavior set for `modelA` using `collision.resolve`.

### See also

`collisionData`, `registerScript()`, `resolve`, `modelA`, `setCollisionCallback()`

## resolveB

### Syntax

```
collisionData.resolveB(bResolve)
```

### Description

3D collision method; overrides the collision behavior set by the `collision.resolve` property for `collisionData.modelB`. If *bResolve* is `TRUE`, then the collision for `modelB` is resolved, if *bResolve* is `FALSE` the collision for the `modelB` is not resolved. Call this function only if you wish to override the behavior set for `modelB` using `collision.resolve`.

### See also

`collisionData`, `resolve`, `registerScript()`, `modelB`, `setCollisionCallback()`

## resource

### Syntax

```
member(whichCastmember).model(whichModel).resource
```

### Description

3D property; allows you to get or set the resource property that defines the geometry of the referenced model resource. This property also allows access to the referenced model's resource object and its associated properties.

### Example

The following statement sets the model resource used by the model named NewBox. It will now have the same geometry as the model named box.

```
member("3d World").model("NewBox").resource = member\  
  ("3d World").model("box").resource
```

This statement displays the resolution property of the model resource used by the model named Cylinder.

```
put member("3d World").model("Cylinder").resource.resolution  
-- 20
```

## restart

### Syntax

```
restart
```

### Description

Command; closes all open applications and restarts the computer.

### Example

This statement restarts the computer when the user presses Command+R (Macintosh) or Control+R (Windows):

```
if the key = "r" and the commandDown then restart
```

### See also

quit, shutDown

## result

### Syntax

```
the result
```

### Description

Function; displays the value of the return expression from the last handler executed.

The `result` function is useful for obtaining values from movies that are playing in windows and tracking Lingo's progress by displaying results of handlers in the Message window as the movie plays.

This function has no effect in the Director player for Java.

To return a result from a handler, assign the result to a variable and then check the variable's value. Use a statement such as `set myVariable = function()`, where *function()* is the name of a specific function.

### Examples

This handler returns a random roll for two dice:

```
on diceRoll
    return random(6) + random(6)
end
```

In the following example, the two statements

```
diceRoll
roll = the result
```

are equivalent to this statement:

```
set roll = diceRoll()
```

Note that `set roll = diceRoll` would not call the handler because there are no parentheses following `diceRoll`; `diceRoll` here is considered a variable reference.

### See also

`return` (keyword)

## resume sprite

### Syntax

```
sprite(whichGIFSpriteNumber ).resume()
resume(sprite whichGIFSpriteNumber)
```

### Description

Animated GIF command; causes the sprite to resume playing from the frame after the current frame if it's been paused. This command has no effect if the animated GIF sprite has not been paused.

### See also

`pause sprite`, `rewind sprite`

## RETURN (constant)

### Syntax

RETURN

### Description

Constant; represents a carriage return.

### Examples

This statement causes a paused movie to continue when the user presses the carriage return:

```
if (the key = RETURN) then go to the frame + 1
```

This statement uses the RETURN character constant to insert a carriage return between two lines in an alert message:

```
alert "Last line in the file." & RETURN & "Click OK to exit."
```

In Windows, it is standard practice to place an additional line-feed character at the end of each line. This statement creates a two-character string named CRLF that provides the additional line feed:

```
CRLF = RETURN & numToChar(10)
```

## return (keyword)

### Syntax

`return expression`

### Description

Keyword; returns the value of *expression* and exits from the handler. The *expression* argument can be any Lingo value.

When calling a handler that serves as a user-defined function and has a return value, you must use parentheses around the argument lists, even if there are no arguments, as in the `diceRoll` function handler discussed under the entry for the `result` function.

The function of the `return` keyword is similar to that of the `exit` command, except that `return` also returns a value to whatever called the handler. The `return` command in a handler immediately exits from that handler, but it can return a value to the Lingo that called it.

The use of `return` in object-oriented scripting can be difficult to understand. It's easier to start by using `return` to create functions and exit handlers. Later, you will see that the `return me` line in an on new handler gives you a way to pass back a reference to an object that was created so it can be assigned to a variable name.

The `return` keyword isn't the same as the character constant `RETURN`, which indicates a carriage return. The function depends on the context.

To retrieve a returned value, use parentheses after the handler name in the calling statement to indicate that the named handler is a function.

To see an example of `return` (keyword) used in a completed movie, see the Parent Scripts movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This handler returns a random multiple of 5 between 5 and 100:

```
on getRandomScore
    theScore = 5 * random(20)
    return theScore
end getRandomScore
```

Call this handler with a statement similar to the following:

```
set thisScore to GetRandomScore()
```

In this example, the variable `thisScore` is assigned the return value from the function `GetRandomScore`. A parent script performs the same function: by returning the object reference, the variable name in the calling code provides a handle for subsequent references to that object.

### See also

`result`, `RETURN` (constant)

## revertToWorldDefaults

### Syntax

```
member(whichCastmember).revertToWorldDefaults()
```

### Description

3D command; reverts the properties of the specified 3D cast member to the values stored when the member was first created. The member's `state` property must be 4 (loaded) or -1 (error) before this command can be used, otherwise a script error will occur.

This command differs from `resetWorld` in that the values used are taken from the state of the member when it was first created rather than from the state of the member when it was first loaded into memory.

### Example

This statement reverts the properties of the cast member named Scene to the values stored when the member was first created:

```
member("Scene").revertToWorldDefaults()
```

### See also

`resetWorld`

## rewind()

### Syntax

```
sound(channelNum).rewind()  
rewind(sound(channelNum))
```

### Description

Function; interrupts the playback of the current sound in sound channel `channelNum` and restarts it at its `startTime`. If the sound is paused, it remains paused, with the `currentTime` set to the `startTime`.

### Example

This statement restarts playback of the sound cast member playing in sound channel 1 from the beginning:

```
sound(1).rewind()
```

### See also

`pause()` (sound playback), `play()` (sound), `playNext()`, `queue()`, `stop()` (sound)

## rewind sprite

### Syntax

```
sprite(whichFlashOrGIFAnimSprite).rewind()  
rewind sprite whichFlashOrGIFAnimSprite
```

### Description

Command; returns a Flash or animated GIF movie sprite to frame 1 when the sprite is stopped or when it is playing.

**Example**

The following frame script checks whether the Flash movie sprite in the sprite the behavior was placed in is playing and, if so, continues to loop in the current frame. When the movie is finished, the sprite rewinds the movie (so the first frame of the movie appears on the Stage) and lets the playhead continue to the next frame.

```
property spriteNum

on exitFrame
  if sprite(spriteNum).playing then
    go the frame
  else
    sprite(spriteNum).rewind()
    updateStage
  end if
end
```

## rgb()

**Syntax**

```
rgb(redValue, greenValue, blueValue)
```

**Description**

Function and data type; defines a color based on a the value specified for red, green, and blue. The range for each of the three color values is 0 - 255.

**Example**

This Lingo displays the color of sprite 6 in the Message window, and then sets the color of sprite 6 to a new RGB value:

```
put sprite(6).color
-- rgb( 255, 204, 102 )
sprite(6).color = rgb(122, 98, 210)
```

**See also**

```
color()
```

## right

**Syntax**

```
sprite(whichSprite).right
the right of sprite whichSprite
```

**Description**

Sprite property; indicates the distance, in pixels, of the specified sprite's right edge from the left edge of the Stage.

When a movie plays back as an applet, this property's value is relative to the upper left corner of the applet.

Sprite coordinates are expressed relative to the upper left corner of the Stage.

This property can be tested and set.

### Example

This statement calls the handler `offRightEdge` when the right edge of sprite 3 is past the right edge of the Stage:

```
if sprite(3).right > (the stageRight - the stageLeft) then offRightEdge
```

### See also

`bottom`, `height`, `left`, `locH`, `locV`, `top`, `width`

## right (3D)

### Syntax

```
member(whichCastmember).modelResource  
(whichModelResource).right  
modelResourceObjectReference.right
```

### Description

3D property; allows you to get or set the `right` property of a model resource whose type is `#box`.

The `right` property determines whether the right of the box is sealed (TRUE) or open (FALSE). The default value is TRUE.

### Example

This statement sets the `right` property of the model resource `Crate` to TRUE, meaning the right side of this box will be closed:

```
member("3D World").modelResource("crate").right = TRUE
```

### See also

`bottom (3D)`, `left (3D)`, `top (3D)`

## rightIndent

### Syntax

```
chunkExpression.rightIndent
```

### Description

Text cast member property; contains the offset distance, in pixels, of the right margin of *chunkExpression* from the right side of the text cast member.

The value is an integer greater than or equal to 0.

This property can be tested and set.

### See also

`firstIndent`, `leftIndent`

## on rightMouseDown (event handler)

### Syntax

```
on rightMouseDown  
  statement(s)  
end
```

**Description**

System message and event handler; in Windows, specifies statements that run when the right mouse button is pressed. On Macintosh computers, the statements run when the mouse button and Control key are pressed simultaneously and the `emulateMultiButtonMouse` property is set to TRUE; if this property is set to FALSE, this event handler has no effect on the Macintosh.

**Example**

This handler opens the window Help when the user clicks the right mouse button in Windows:

```
on rightMouseDown
    window("Help").open()
end
```

## rightMouseDown (system property)

**Syntax**

```
the rightMouseDown
```

**Description**

System property; indicates whether the right mouse button (Windows) or the mouse button and Control key (Macintosh) are being pressed (TRUE) or not (FALSE).

On the Macintosh, `rightMouseDown` is TRUE only if the `emulateMultiButtonMouse` property is TRUE.

**Example**

This statement checks whether the right mouse button in Windows is being pressed and plays the sound Oops in sound channel 2 if it is:

```
if the rightMouseDown then puppetSound 2, "Oops"
```

**See also**

`emulateMultiButtonMouse`

## on rightMouseUp (event handler)

**Syntax**

```
on rightMouseUp
    statement(s)
end
```

**Description**

System message and event handler; in Windows, specifies statements that run when the right mouse button is released. On Macintosh computers, the statements run if the mouse button is released while the Control key is pressed and the `emulateMultiButtonMouse` property is set to TRUE; if this property is set to FALSE, this event handler has no effect on the Macintosh.

**Example**

This handler opens the Help window when the user releases the right mouse button in Windows:

```
on rightMouseUp
    window("Help").open()
end
```



## rightMouseUp (system property)

### Syntax

the rightMouseUp

### Description

System property; indicates whether the right mouse button (Windows) or the mouse button and Control key (Macintosh) are currently not being pressed (TRUE) or are currently being pressed (FALSE).

On the Macintosh, rightMouseUp is TRUE only if the emulateMultiButtonMouse property is TRUE.

### Example

This statement checks whether the right mouse button in Windows is released and plays the sound Click Me if it is:

```
if the rightMouseUp then puppetSound 2, "Click Me"
```

### See also

emulateMultiButtonMouse

# rollOver()

## Syntax

`rollOver(whichSprite)`  
the `rollOver`

## Description

Function; indicates whether the pointer (cursor) is currently over the bounding rectangle of the sprite specified by *whichSprite* (TRUE or 1) or not (FALSE or 0).

This function has two possible syntax formats:

- When `rollOver` isn't preceded by `the`, include parentheses.
- When `rollOver` is preceded by `the`, don't include parentheses.

The `rollOver` function is typically used in frame scripts and is useful for creating handlers that perform an action when the user places the pointer over a specific sprite. It can also simulate additional sprite channels by splitting the Stage into regions that each send the playhead to a different frame that subdivides the region for the available sprite channels.

If the user continues to roll the mouse, the value of `rollOver` can change while Lingo is running a handler. You can make sure that a handler uses a consistent rollover value by assigning `rollOver` to a variable when the handler starts.

When the pointer is over the location of a sprite that no longer appears in the Score in the current section, `rollOver` still occurs and reports the sprite as being there. Avoid this problem by not performing rollovers over these locations or by moving the sprite above the menu bar before removing it.

## Examples

This statement changes the content of the field cast member `Message` to "This is the place." when the pointer is over sprite 6:

```
if rollover(6) then member("Message").text = "This is the place."
```

The following handler sends the playhead to different frames when the pointer is over certain sprites on the Stage. It first assigns the `rollOver` value to a variable. This lets the handler use the `rollOver` value that was in effect when the rollover started, regardless of whether the user continues to move the mouse.

```
on exitFrame  
  set currentSprite = the rollover  
  case currentSprite of  
    1: go to frame "Left"  
    2: go to frame "Middle"  
    3: go to frame "Right"  
  end case  
end exitFrame
```

## See also

`mouseMember`

## romanLingo

### Syntax

the romanLingo

### Description

System property; specifies whether Lingo uses a single-byte (TRUE) or double-byte interpreter (FALSE).

The Lingo interpreter is faster with single-byte character sets. Some versions of Macintosh system software—Japanese, for example—use a double-byte character set. U.S. system software uses a single-byte character set. Normally, `romanLingo` is set when Director is first started and is determined by the local version of the system software.

If you are using a non-Roman script system but don't use any double-byte characters in your script, set this property to TRUE for faster execution of your Lingo scripts.

### Example

This statement sets `romanLingo` to TRUE, which causes Lingo to use a single-byte character set:

```
set the romanLingo to TRUE
```

## rootLock

### Syntax

```
member(whichCastmember).model(whichModel).keyframePlayer.rootLock  
member(whichCastmember).model(whichModel).bonesPlayer.rootLock
```

### Description

3D `#keyframePlayer` and `#bonesPlayer` modifier property; indicates whether the translational components of a motion are used (FALSE) or ignored (TRUE).

The default value of this property is FALSE.

### Example

This statement forces the model named Alien3 to remain at its starting position while executing its motions, resulting in a character that will walk in place:

```
member("newalien").model("Alien3").keyframePlayer.rootLock = 1
```

## rootNode

### Syntax

```
member(whichCastmember).camera(whichCamera).rootNode  
sprite(whichSprite).camera.rootNode
```

### Description

3D property; allows you to get or set which objects are visible within a sprite. When a camera is first created, it shows all nodes within the world. The `rootNode` property allows you to modify this by creating a different default view that limits what's shown to a particular node and its children.

For example, light C is a child of model A., if you set the `rootNode` property to `camera("defaultView").rootNode=model(A)`, the sprite will show only model A as illuminated by light C. The default is `group("world")`, meaning that all nodes are used.

### Example

The following statement sets the `rootNode` of the camera of sprite 5 to the model Pluto. Only Pluto and its children will be visible in sprite 5.

```
sprite(5).camera.rootNode = member("Scene").model("Pluto")
```

## rotate

### Syntax

```
member(whichCastmember).node(whichNode).rotate(xAngle, yAngle, \
    zAngle {, relativeTo})
member(whichCastmember).node(whichNode).rotate(rotationVector \
    {, relativeTo})
member(whichCastmember).node(whichNode).rotate(position, axis, \
    angle {, relativeTo})
transform.rotate(xAngle, yAngle, zAngle {, relativeTo})
transform.rotate(rotationVector {, relativeTo})
transform.rotate(position, axis, angle {, relativeTo})
```

### Description

3D command; applies a rotation after the current positional, rotational, and scale offsets held by the node's transform object or the directly referenced transform object. The rotation must be specified as a set of three angles, each of which specify an angle of rotation about the three corresponding axes. These angles may be specified explicitly in the form of *xAngle*, *yAngle*, and *zAngle*, or by a *rotationVector*, where the *x* component of the vector corresponds to the rotation about the X axis, *y* about Y axis, and *z* about Z axis. Alternatively, the rotation may also be specified as a rotation about an arbitrary axis passing through a point in space. This axis is defined in space by *position*, representing a position in space and *axis*, representing an axis passing through the specified position in space. The amount of rotation about this axis is specified by *angle*.

The optional *relativeTo* parameter determines which coordinate system axes are used to apply the desired rotational changes. The *relativeTo* parameter can have any of the following values:

- *#self* applies the increments relative to the node's local coordinate system (the X, Y and Z axes specified for the model during authoring). This value is used as the default if you use the `rotate` command with a node reference and the *relativeTo* parameter is not specified.
- *#parent* applies the increments relative to the node's parent's coordinate system. This value is used as the default if you use the `rotate` command with a transform reference and the *relativeTo* parameter is not specified.
- *#world* applies the increments relative to the world coordinate system. If a model's parent is the world, then this is equivalent to using *#parent*.
- *nodeReference* allows you to specify a node to base your rotation upon, the command applies the increments relative to the coordinate system of the specified node.

### Examples

The following example first rotates the model named Moon about its own Z axis (rotating it in place), then it rotates that same model about its parent node, the model named Earth (causing Moon to move orbitally about Earth).

```
member("Scene").model("Moon").rotate(0,0,15)
member("Scene").model("Moon").rotate(vector(005)member("Scene").model("Moon"))
```

The following example rotates the model Ball around a position in space occupied by the model named Pole. The effect is that the model Ball moves orbitally about Pole in the x-y plane.

```
polePos = member("3d Scene").model("Pole").worldPosition
member("3d Scene").model("Ball").rotate(polePos, vector(0,0,1), \
5, #world)
```

#### See also

`pointAt`, `preRotate`, `rotation (transform)`, `rotation (engraver shader)`, `rotation (backdrop and overlay)`, `preScale()`, `transform (property)`

## rotation

### Syntax

the rotation of member *whichQuickTimeMember*  
`member(whichQuickTimeMember).rotation`  
sprite(*whichSprite*).rotation  
the rotation of sprite *whichSprite*

### Description

Cast member property and sprite property; controls the rotation of a QuickTime movie, animated GIF, Flash movie, or bitmap sprite within the sprite's bounding rectangle, without rotating that rectangle or the sprite's controller (in the case of QuickTime). In effect, the sprite's bounding rectangle acts as a window through which you can see the Flash or QuickTime movie. The bounding rectangles of bitmaps and animated GIFs change to accommodate the rotating image.

Score rotation works for a Flash movie only if `obeyScoreRotation` is set to `TRUE`.

A Flash movie rotates around its origin point as specified by its `originMode` property. A QuickTime movie rotates around the center of the bounding rectangle of the sprite. A bitmap rotates around the registration point of the image.

For QuickTime media, if the sprite's `crop` property is set to `TRUE`, rotating the sprite frequently moves part of the image out of the viewable area; when the sprite's `crop` property is set to `FALSE`, the image is scaled to fit within the bounding rectangle (which may cause image distortion).

You specify the rotation in degrees as a floating-point number.

The Score can retain information for rotating an image from +21,474,836.47° to -21,474,836.48°, allowing 59,652 full rotations in either direction.

When the rotation limit is reached (slightly past the 59,652th rotation), the rotation resets to +116.47° or -116.48°—not 0.00°. This is because +21,474,836.47° is equal to +116.47°, and -21,474,836.48° is equal to -116.48° (or +243.12°). To avoid this reset condition, when you use Lingo to perform continuous rotation, constrain the angles to ±360°.

This property can be tested and set. The default value is 0.

### Examples

This behavior causes a sprite to rotate continuously by 2° every time the playhead advances, limiting the angle to 360°:

```
property spriteNum
```

```
on prepareFrame me
    sprite(spriteNum).rotation = integer(sprite(spriteNum).rotation + 2) mod 360
end
```

The following frame script keeps the playhead looping in the current frame while it rotates a QuickTime sprite in channel 5 a full 360° in 16° increments. When the sprite has been rotated 360°, the playhead continues to the next frame.

```
on exitFrame
  if sprite(5).rotation < 360 then
    sprite(5).rotation = sprite(5).rotation + 16
    go the frame
  end if
end
```

This handler accepts a sprite reference as a parameter and rotates a Flash movie sprite 360° in 10° increments:

```
on rotateMovie whichSprite
  repeat with i = 1 to 36
    sprite(whichSprite).rotation = i * 10
    updatestage
  end repeat
end
```

**See also**

flipH, flipV, obeyScoreRotation, originMode

## rotation (backdrop and overlay)

**Syntax**

```
sprite(whichSprite).camera.backdrop[backdropIndex].rotation
member(whichCastmember).camera(whichCamera).backdrop
[backdropIndex].rotation
sprite(whichSprite).camera.overlay[overlayIndex].rotation
member(whichCastmember).camera[cameraIndex].overlay
[overlayIndex].rotation
```

**Description**

3D property; allows you to get or set the rotation of the backdrop or overlay toward the default camera. The default value of this property is 0.0.

**Example**

This statement rotates a backdrop 60° around its registration point:

```
sprite(4).camera.backdrop[1].rotation = 60.0
```

**See also**

bevelDepth, transform (property)

## rotation (engraver shader)

**Syntax**

```
member(whichCastmember).shader(whichShader).rotation
member(whichCastmember).model(whichModel).shader.rotation
member(whichCastmember).model(whichModel).shaderList[index].rotation
```

**Description**

3D shader engraver property; allows you to get or set an angle in degrees (as a floating-point number) that describes a 2D rotational offset for engraved lines. The default value for this property is 0.0.

### Example

This statement rotates the lines used to draw the engraver shader for the model gbCyl3 by 1°:

```
member("scene").model("gbCyl3").shader.rotation = \  
    member("scene").model("gbCyl3").shader.rotation + 1
```

### See also

`transform (property)`

## rotation (transform)

### Syntax

```
member(whichCastmember).node(whichNode).transform.rotation  
member(whichCastmember).node(whichNode).getWorldTransform().rotation  
transform.rotation
```

### Description

3D property; allows you to get or set the rotational component of a transform. A transform defines a scale, position and rotation within a given frame of reference. The default value of this property is `vector(0,0,0)`.

A node can be a camera, group, light or model object. Setting the `rotation` of a node's transform defines that object's rotation within the transform's frame of reference. Setting the `rotation` property of an object's world relative transform using `getWorldTransform().rotation` defines the object's rotation relative to the world origin. Setting the `rotation` property of an object's parent relative transform using `transform.rotation` defines the object's rotation relative to its parent node.

If you wish to modify the orientation of a transform it is recommended that you use the `rotate` and `prerotate` methods instead of setting this property.

### Examples

This statement sets the parent-relative rotation of the first camera in the member to `vector(0,0,0)`:

```
member("Space").camera[1].transform.rotation = vector(0, 0, 0)
```

This example displays the parent-relative rotation of the model named Moon, then it adjusts the model's orientation using the `rotate` command, and finally it displays the resulting world-relative rotation of the model:

```
put member("SolarSys").model("Moon").transform.rotation  
-- vector( 0.0000, 0.0000, 45.0000)  
member("SolarSys").model("Moon").rotate(15,15,15)  
put member("SolarSys").model("Moon").getWorldTransform().rotation  
--vector( 51.3810, 16.5191, 65.8771 )
```

### See also

`getWorldTransform()`, `preRotate`, `rotate`, `transform (property)`, `position (transform)`, `scale (transform)`

## rotationReset

### Syntax

```
member(whichCastmember).model(whichModel).bonesPlayer.rotationReset  
member(whichCastmember).model(whichModel).keyframePlayer.\  
    rotationReset
```

### Description

3D keyframePlayer and bonesPlayer modifier property; indicates the axes around which rotational changes are maintained from the end of one motion to the beginning of the next, or from the end of one iteration of a looped motion to the beginning of the next iteration.

Possible values of this property include #none, #x, #y, #z, #xy, #yz, #xz, and #all. The default value is #all.

### Example

This statement sets the rotationReset property of the model named Monster to the z-axis. The model maintains rotation around its z-axis when the currently playing motion or loop ends.

```
member("NewAlien").model("Monster").bonesPlayer.rotationReset = #z
```

### See also

positionReset, bonesPlayer (modifier)

## RTF

### Syntax

```
member(whichMember).RTF
```

### Description

Cast member property; allows access to the text and tags that control the layout of the text within a text cast member containing text in rich text format.

This property can be tested and set.

### Example

This statement displays in the Message window the RTF formatting information embedded in the text cast member Resume:

```
put member("Resume").RTF
```

### See also

HTML, importFileInto

## runMode

### Syntax

```
the runMode
```

### Description

Function; returns a string indicating the mode in which the movie is playing. Possible values are as follows:

- Author—The movie is running in Director.
- Projector—The movie is running as a projector.



- **BrowserPlugin**—The movie is running as a Shockwave plug-in or other scripting environment, such as LiveConnect or ActiveX.
- **Java Applet**—The movie is playing back as a Java applet.

The safest way to test for particular values in this property is to use the `contains` operator. This helps avoid errors and allows partial matches.

### Example

This statement determines whether or not external parameters are available and obtains them if they are:

```
if the runMode contains "Plugin" then
  -- decode the embed parameter
  if externalParamName(swURL) = swURL then
    put externalParamValue(swURL) into myVariable
  end if
end if
```

### See also

environment, platform

## on runPropertyDialog

### Syntax

```
on runPropertyDialog me, currentInitializerList
  statement(s)
end
```

### Description

System message and event handler; contains Lingo that defines specific values for a behavior's parameters in the Parameters dialog box. The `runPropertyDialog` message is sent whenever the behavior is attached to a sprite, or when the user changes the initial property values of a sprite's behavior.

The current settings for a behavior's initial properties are passed to the handler as a property list. If the `on runPropertyDialog` handler is not defined within the behavior, Director runs a behavior customization dialog box based on the property list returned by the `on getPropertyDescriptionList` handler.

### Example

The following handler overrides the behavior's values set in the Parameters dialog box for the behavior. New values are contained in the list `currentInitializerList`. Normally, the Parameters dialog box allows the user to set the mass and gravitational constants. However, this handler assigns these parameters constant values without displaying a dialog box:

```
property mass
property gravitationalConstant
on runPropertyDialog me, currentInitializerList
  --force mass to 10
  setaProp currentInitializerList, #mass, 10
  -- force gravitationalConstant to 9.8
  setaProp currentInitializerList, #gravitationalConstant, 9.8
  return currentInitializerList
end
```

### See also

on `getBehaviorDescription`, on `getPropertyDescriptionList`

## safePlayer

### Syntax

the `safePlayer`

### Description

System property; controls whether or not safety features in Director are turned on.

In a Shockwave movie, this property can be tested but not set. It is always `TRUE` in Shockwave.

In the authoring environment and in projectors, the default value is `FALSE`. This property may be tested, but it may only be set to `TRUE`. Once it has been set to `TRUE`, it cannot be set back to `FALSE` without restarting Director or the projector.

When `safePlayer` is `TRUE`, the following safety features are in effect:

- Only safe Xtra extensions may be used.
- The `safePlayer` property cannot be reset.
- Pasting content from the Clipboard by using the `pasteClipboardInto` command generates a warning dialog box that allows the user to cancel the operation.
- Handling Macintosh resource files by using the obsolete `openResFile` or `closeResFile` command is disabled.
- Saving a movie or cast by using Lingo is disabled.
- Printing by using the `printFrom` command is disabled.
- Opening an application by using the `open` command is disabled.
- The ability to stop an application or the user's computer by using the `restart` or `shutDown` command is disabled.
- Sending strings to Windows Media Control Interface (MCI) by using `mci` is disabled.
- Opening a file that is outside the `DSWMedia` folder is disabled.
- Discovering a local filename is disabled.
- Using `getNetText()` or `postNetText()`, or otherwise accessing a URL that does not have the same domain as the movie, generates a security dialog box.

## sampleCount

### Syntax

`sound(channelNum).sampleCount`  
the `sampleCount` of `sound(channelNum)`

### Description

Read-only property; the number of sound samples in the currently playing sound in sound channel `channelNum`. This is the total number of samples, and depends on the `sampleRate` and `duration` of the sound. It does not depend on the `channelCount` of the sound.

A 1-second, 44.1 KHz sound contains 44,100 samples.

### Example

This statement displays the name and `sampleCount` of the cast member currently playing in sound channel 1 in the Message window:

```
put "Sound cast member" && sound(1).member.name && "contains" && \
sound(1).sampleCount && "samples."
```

### See also

`channelCount`, `sampleRate`, `sampleSize`

## sampleRate

### Syntax

`member(whichCastMember).sampleRate`  
the `sampleRate` of member *whichCastMember*  
`sound(channelNum).sampleRate`

### Description

Cast member property; returns, in samples per second, the sample rate of the sound cast member or in the case of SWA sound, the original file that has been Shockwave Audio–encoded. This property is available only after the SWA sound begins playing or after the file has been preloaded using the `preLoadBuffer` command. When a sound channel is given, the result is the sample rate of the currently playing sound cast member in the given sound channel.

This property can be tested but not set. Typical values are 8000, 11025, 16000, 22050, and 44100.

When multiple sounds are queued in a sound channel, Director plays them all with the `channelCount`, `sampleRate`, and `sampleSize` of the first sound queued, resampling the rest for smooth playback. Director resets these properties only after the channel's sound queue is exhausted or a `stop()` command is issued. The next sound to be queued or played then determines the new settings.

### Examples

This statement assigns the original sample rate of the file used in SWA streaming cast member Paul Robeson to the field cast member Sound Quality:

```
member("Sound Quality").text = string(member("Paul Robeson").sampleRate)"
```

This statement displays the sample rate of the sound playing in sound channel 1 in the Message window:

```
put sound(1).sampleRate
```

## sampleSize

### Syntax

`member(whichCastMember).sampleSize`  
the `sampleSize` of member *whichCastMember*  
`sound(channelNum).sampleSize`

### Description

Cast member property; determines the sample size of the specified cast member. The result is usually a size of 8 or 16 bits. If a sound channel is given, the value if for the sound member currently playing in the given sound channel.

This property can be tested but not set.

### Examples

This statement checks the sample size of the sound cast member Voice Over and assigns the value to the variable `soundSize`:

```
soundSize = member("Voice Over").sampleSize
```

This statement displays the sample size of the sound playing in sound channel 1 in the Message window:

```
put sound(1).sampleSize
```

## save castLib

### Syntax

```
castLib(whichCast).save()  
save castLib whichCast {,pathName&newFileName}
```

### Description

Command; saves changes to the cast in the cast's original file or, if the optional parameter *pathName:newFileName* is included, in a new file. If no filename is given, the original cast must be linked. Further operations or references to the cast use the saved cast member.

This command does not work with compressed files.

The `save CastLib` command doesn't support URLs as file references.

### Example

This statement causes Director to save the revised version of the Buttons cast in the new file `UpdatedButtons` in the same folder:

```
castLib("Buttons").save(the moviePath & "UpdatedButtons.cst")
```

### See also

@ (pathname)

## on savedLocal

### Syntax

```
on savedLocal  
    statement(s)  
end
```

### Description

System message and event handler; This property is provided to allow for enhancements in future versions of Shockwave.

### See also

allowSaveLocal

## saveMovie

### Syntax

```
saveMovie {pathName&fileName}
```

### Description

Command; saves the current movie. Including the optional parameter saves the movie to the file specified by *pathName:fileName*. This command does not work with compressed files. The specified filename must include the .dir file extension.

The `saveMovie` command doesn't support URLs as file references.

### Example

This statement saves the current movie in the Update file:

```
saveMovie the moviePath & "Update.dir"
```

### See also

@ (pathname), setPref

## scale

### Syntax

```
member(whichCastMember).scale  
the scale of member whichCastMember  
sprite(whichSprite).scale  
the scale of sprite whichSprite
```

### Description

Cast member property and sprite property; controls the scaling of a QuickTime, vector shape, or Flash movie sprite.

For QuickTime, this property does not scale the sprite's bounding rectangle or the sprite's controller. Instead, it scales the image around the image's center point within the bounding rectangle. The scaling is specified as a Director list containing two percentages stored as float-point values:

```
[xPercent, yPercent]
```

The *xPercent* parameter specifies the amount of horizontal scaling; the *yPercent* parameter specifies vertical scaling.

When the sprite's `crop` property is set to `TRUE`, the `scale` property can be used to simulate zooming within the sprite's bounding rectangle. When the sprite's `crop` property is set to `FALSE`, the `scale` property is ignored.

This property can be tested and set. The default value is [1.0000,1.0000].

For Flash movie or vector shape cast members, the `scale` is a floating-point value. The movie is scaled from its origin point, as specified by its `originMode` property.

**Note:** This property must be set to the default value if the `scaleMode` property is set to `#autoSize`; otherwise the sprite does not display correctly.

### Examples

The following sprite script keeps the playhead looping in the current frame while the QuickTime sprite in channel 5 is scaled down in 5% increments. When the sprite is no longer visible (because its horizontal scale value is 0% or less), the playhead continues to the next frame.

```
on exitFrame me
    scaleFactor = sprite(spriteNum).scale[1]
    currentMemberNum = sprite(spriteNum).memberNum
    if member(currentMemberNum).crop = FALSE then
        member(currentMemberNum).crop = TRUE
    end if
    if scaleFactor > 0 then
        scaleFactor = scaleFactor - 5
        sprite(spriteNum).scale = [scaleFactor, scaleFactor]
        go the frame
    end if
end
```

This handler accepts a reference to a Flash movie sprite as a parameter, reduces the movie's scale to 0% (so it disappears), and then scales it up again in 5% increments until it is full size (100%) again:

```
on scaleMovie whichSprite
    sprite(whichSprite).scale = 0
    updatestage
    repeat with i = 1 to 20
        sprite(whichSprite).scale = i * 5
        updatestage
    end repeat
end
```

### See also

scaleMode, originMode

## scale (backdrop and overlay)

### Syntax

```
member(whichCastmember).camera(whichCamera).backdrop\  
    [backdropIndex].scale  
member(whichCastmember).camera(whichCamera).overlay\  
    [overlayIndex].scale
```

### Description

3D property; allows you to get or set the scale value used by a specific overlay or backdrop in the referenced camera's list of overlays or backdrops to display. The width and height of the backdrop or overlay are multiplied by the scale value. The default value for this property is 1.0.

### Example

This statement doubles the size of a backdrop:

```
sprite(25).camera.backdrop[1].scale = 2.0
```

### See also

bevelDepth, overlay

## scale (command)

### Syntax

```
member(whichCastmember).node(whichNode).scale(xScale, yScale, \
zScale)
member(whichCastmember).node(whichNode).scale(uniformScale)
transform.scale(xScale, yScale, zScale)
transform.scale(uniformScale)
```

### Description

3D transform command; applies a scaling after the current positional, rotational, and scale offsets held by a referenced node's transform or the directly referenced transform. The scaling must be specified as either a set of three scalings along the corresponding axes or as a single scaling to be applied uniformly along all axes. You can specify the individual scalings using the *xScale*, *yScale* and *zScale* parameters, otherwise you can specify the uniform scaling amount using the *uniformScale* parameter.

A node can be a camera, group, light or model object. Using the `scale` command adjusts the referenced node's `transform.scale` property, but it does not have any visual effect on lights or cameras as they do not contain geometry.

The scaling values provided must be greater than zero.

### Examples

This example first displays the `transform.scale` property for the model named Moon, then it scales the model using the `scale` command, and finally, it displays the resulting `transform.scale` value.

```
put member("Scene").model("Moon").transform.scale
-- vector( 1.0000, 1.0000, 1.0000)
member("Scene").model("Moon").scale(2.0,1.0,0.5)
put member("Scene").model("Moon").transform.scale
-- vector( 2.0000, 1.0000, 0.5000)
```

This statement scales the model named Pluto uniformly along all three axes by 0.5, resulting in the model displaying at half of its size.

```
member("Scene").model("Pluto").scale(0.5)
```

This statement scales the model named Oval in a nonuniform manner, scaling it along its *z*-axis but not its *x*- or *y*-axes.

```
member("Scene").model("Pluto").scale(0.0, 0.0, 0.5)
```

### See also

```
transform (property), preScale(), scale (transform)
```

## scale (transform)

### Syntax

```
member(whichCastmember).node(whichNode).transform.scale  
member(whichCastmember).node(whichNode).getWorldTransform().scale  
transform.scale
```

### Description

3D property; allows you to get or set the scaling component of a transform. A transform defines a scale, position and rotation within a given frame of reference. The `scale` property allows you to get and set the degree of scaling of the transform along each of the three axes. The default value of this property is `vector(1.0,1.0,1.0)`.

A node can be a camera, group, light or model object. This command does not have any visual effect on lights or cameras as they do not contain geometry. Setting the `scale` property of a node's transform defines that object's scaling along the X, Y and Z axes within the transform's frame of reference. Getting the `scale` property of an object's world relative transform using `getWorldTransform().scale` returns the object's scaling relative to the world origin. Setting the `scale` property of an object's parent relative transform using `transform.scale` defines the object's scaling relative to its parent node.

### Example

This statement sets the `scale` property of the transform of the model named Moon to `vector(2,5,3)`:

```
member("Scene").model("Moon").transform.scale = vector(2,5,3)
```

### See also

`transform (property)`, `getWorldTransform()`, `position (transform)`, `rotation (transform)`, `scale (command)`

## scaleMode

### Syntax

```
sprite(whichVectorOrFlashSprite).scaleMode  
the scaleMode of sprite whichVectorOrFlashSprite  
member(whichVectorOrFlashMember).scaleMode  
the scaleMode of member whichVectorOrFlashMember
```

### Description

Cast member property and sprite property; controls the way a Flash movie or vector shape is scaled within a sprite's bounding rectangle. When you scale a Flash movie sprite by setting its `scale` and `viewScale` properties, the sprite itself is not scaled; only the view of the movie within the sprite is scaled. The `scaleMode` property can have these values:

- `#showAll` (default for Director movies prior to version 7)—Maintains the aspect ratio of the original Flash movie cast member. If necessary, fill in any gap in the horizontal or vertical dimension using the background color.
- `#noBorder`—Maintains the aspect ratio of the original Flash movie cast member. If necessary, crop the horizontal or vertical dimension.
- `#exactFit`—Does not maintain the aspect ratio of the original Flash movie cast member. Stretch the Flash movie to fit the exact dimensions of the sprite.



- `#noScale`—preserves the original size of the Flash media, regardless of how the sprite is sized on the Stage. If the sprite is made smaller than the original Flash movie, the movie displayed in the sprite is cropped to fit the bounds of the sprite.
- `#autoSize` (default)—This specifies that the sprite rectangle is automatically sized and positioned to account for rotation, skew, `flipH`, and `flipV`. This means that when a Flash sprite is rotated, it will not crop as in earlier versions of Director. The `#autoSize` setting only functions properly when `scale`, `viewScale`, `originPoint`, and `viewPoint` are at their default values.

This property can be tested and set.

#### Example

The following sprite script checks the Stage color of the Director movie and, if the Stage color is indexed to position 0 in the current palette, the script sets the `scaleMode` property of a Flash movie sprite to `#showAll`. Otherwise, it sets the `scaleMode` property to `#noBorder`.

```
on beginsprite me
  if the stagecolor = 0 then
    sprite(me.spriteNum).scaleMode = #showAll
  else
    sprite(me.spriteNum).scaleMode = #noBorder
  end if
end
```

#### See also

`scale`

## score

#### Syntax

the score

#### Description

Movie property; determines which Score is associated with the current movie. This property can be useful for storing the current contents of the Score before wiping out and generating a new one or for assigning the current Score contents to a film loop.

This property can be tested and set.

#### Example

This statement assigns the film loop cast member `Waterfall` to the Score of the current movie:

```
the score = member("Waterfall").media
```

## scoreColor

### Syntax

`sprite(whichSprite).scoreColor`  
the `scoreColor` of sprite *whichSprite*

### Description

Sprite property; indicates the Score color assigned to the sprite specified by *whichSprite*. The possible values correspond to color chips 0 to 5 in the current palette.

This property can be tested and set. Setting this property is useful only during authoring and Score recording.

### Example

This statement displays in the Message window the value for the Score color assigned to sprite 7:

```
put sprite(7).scorecolor
```

## scoreSelection

### Syntax

the `scoreSelection`

### Description

Movie property; determines which channels are selected in the Score window. The information is formatted as a linear list of linear lists. Each contiguous selection is in a list format consisting of the starting channel number, ending channel number, starting frame number, and ending frame number. Specify sprite channels by their channel numbers; use the following numbers to specify the other channels.

To specify:	Use:
Frame script channel	0
Sound channel 1	-1
Sound channel 2	-2
Transition channel	-3
Palette channel	-4
Tempo channel	-5

You can select discontinuous channels or frames.

This property can be tested and set.

### Examples

This statement selects sprite channels 15 through 25 in frames 100 through 200:

```
set the scoreSelection = [[15, 25, 100, 200]]
```

This statement selects sprite channels 15 through 25 and 40 through 50 in frames 100 through 200:

```
set the scoreSelection = [[15, 25, 100, 200], [40, 50, 100, 200]]
```

This statement selects the frame script in frames 100 through 200:

```
set the scoreSelection = [[0, 0, 100, 200]]
```

# script

## Syntax

the script of menuItem *whichItem* of menu *whichMenu*  
*childObject*.script  
the script of *childObject*

## Description

Menu item and child object property.

For menu items, determines which Lingo statement is executed when the specified menu item is selected. The *whichItem* expression can be either a menu item name or a menu item number; the *whichMenu* expression can be either a menu name or a menu number.

When a menu is installed, the script is set to the text following the “Å” character in the menu definition.

This property can be tested and set.

**Note:** Menus are not available in Shockwave.

For child objects, the return value is the name of the child’s parent script. This property can be tested but not set.

To see an example of `script` used in a completed movie, see the Parent Scripts movie in the Learning/Lingo Examples folder inside the Director application folder.

## Examples

This statement makes `goHandler` the handler that is executed when the user chooses the Go command from the custom menu Control:

```
set the script of menuItem "Go" of menu "Control" to "goHandler"
```

This Lingo checks whether a child object is an instance of the parent script Warrior Ant:

```
if bugObject.script = script("Warrior Ant") then
    bugObject.attack()
end if
```

## See also

`checkMark`, `installMenu`, `menu`, `handlers()`, `scriptText`

# scriptInstanceList

## Syntax

sprite(*whichSprite*).scriptInstanceList  
the scriptInstanceList of sprite *whichSprite*

## Description

Sprite property; creates a list of script references attached to a sprite. This property is available only during run time. The list is empty when the movie is not running. Modifications to the list are not saved in the Score. This property is useful for the following tasks:

- Attaching a behavior to a sprite for use during run time
- Determining if behaviors are attached to a sprite; determining what the behaviors are
- Finding a behavior script reference to use with the `sendSprite` command

This property can be tested and set. (It can be set only if the sprite already exists and has at least one instance of a behavior already attached to it.)

### Examples

This handler displays the list of script references attached to a sprite:

```
on showScriptRefs spriteNum
  put sprite(spriteNum).scriptInstanceList
end
```

These statements attach the script Big Noise to sprite 5:

```
x = script("Big Noise").new()
sprite(5).scriptInstanceList.add(x)
```

### See also

scriptNum, sendSprite

## scriptList

### Syntax

sprite(*whichSprite*).scriptList  
the scriptList of sprite *whichSprite*

### Description

Sprite property; returns the list of behaviors attached to the given sprite and their properties. This property may only be set by using `setScriptList()`. It may not be set during a score recording session.

### Example

This statement displays the list of scripts attached to sprite 1 in the Message window:

```
put sprite(1).scriptList
--[[ (member2ofcastLib1), ["#myRotateAngle:10.0000,#myClockwise:1,#myInitialAngle:0.0000"]], (member3ofcastLib1), ["#myAnglePerFrame:10.0000,#myTurnFrames:10,#myHShiftPerFrame:10 #myShiftFrames:10 #myTotalFrames:60 #mySurfaceHeight:0"] ]]
```

### See also

setScriptList(), value()

## scriptNum

### Syntax

sprite(*whichSprite*). scriptNum  
scriptNum of sprite *whichSprite*

### Description

Sprite property; indicates the number of the script attached to the sprite specified by *whichSprite*. If the sprite has multiple scripts attached, `scriptNum` sprite property returns the number of the first script. (To see a complete list of the scripts attached to a sprite, see the behaviors listed for that sprite in the Behavior Inspector.)

This property can be tested and set during Score recording.

### Example

This statement displays the number of the script attached to sprite 4:

```
put sprite(4).scriptNum
```

### See also

`scriptInstanceList`

## scriptsEnabled

### Syntax

`member(whichCastMember).scriptsEnabled`  
the `scriptsEnabled` of member *whichCastMember*

### Description

Director movie cast member property; determines whether scripts in a linked movie are enabled (TRUE or 1) or disabled (FALSE or 0).

This property is available for linked Director movie cast members only.

This property can be tested and set.

### Example

This statement turns off scripts in the linked movie Jazz Chronicle:

```
member("Jazz Chronicle").scriptsEnabled = FALSE
```

## scriptText

### Syntax

`member(whichCastMember).scriptText`  
the `scriptText` of member *whichCastMember*

### Description

Cast member property; indicates the content of the script, if any, assigned to the cast member specified by *whichCastMember*.

The text of a script is removed when a movie is converted to a projector, protected, or compressed for Shockwave. Such movies then lose their values for the `scriptText` cast member property.

Therefore, the movie's `scriptText` cast member property values can't be retrieved when the movie is played back by a projector. However, Director can set new values for `scriptText` cast member property inside the projector. These newly assigned scripts are automatically compiled so that they execute quickly.

This property can be tested and set.

### Example

This statement makes the contents of field cast member 20 the script of cast member 30:

```
member(30).scriptText = member(20).text
```

## scriptType

### Syntax

member *whichScript*.scriptType  
the scriptType of member *whichScript*

### Description

Cast member property; indicates the specified script's type. Possible values are #movie, #score, and #parent.

### Example

This statement makes the script member Main Script a movie script:

```
member("Main Script").scriptType = #movie
```

## scrollByLine

### Syntax

member(*whichCastMember*). scrollByLine(*amount*)  
scrollByLine member *whichCastMember*, *amount*

### Description

Command; scrolls the specified field or text cast member up or down by the number of lines specified in *amount*. (Lines are defined as lines separated by carriage returns or by wrapping.)

- When *amount* is positive, the field scrolls down.
- When *amount* is negative, the field scrolls up.

### Examples

This statement scrolls the field cast member Today's News down five lines:

```
member("Today's News").scrollbyline(5)
```

This statement scrolls the field cast member Today's News up five lines:

```
scrollByLine member "Today's News", -1
```

## scrollByPage

### Syntax

member(*whichCastMember*). scrollByPage(*amount*)  
scrollByPage member *whichCastMember*, *amount*

### Description

Command; scrolls the specified field or text cast member up or down by the number of pages specified in *amount*. A page is equal to the number of lines of text visible on the screen.

- When *amount* is positive, the field scrolls down.
- When *amount* is negative, the field scrolls up.

The Director player for Java doesn't support the scrollByPage member property. Use the scrollTop member property to write Lingo that scrolls text.

## Examples

This statement scrolls the field cast member Today's News down one page:

```
member("Today's News").scrollbypage(1)
```

This statement scrolls the field cast member Today's News up one page:

```
member("Today's News").scrollbypage(-1)
```

## See also

`scrollTop`

# scrollTop

## Syntax

```
member(whichCastMember).scrollTop  
the scrollTop of member whichCastMember
```

## Description

Cast member property; determines the distance, in pixels, from the top of a field cast member to the top of the field that is currently visible in the scrolling box. By changing the value for `scrollTop` member property while the movie plays, you can change the section of the field that appears in the scrolling field.

This is a way to make custom scrolling behaviors for text and field members.

For example, the following Lingo moves the field cast member Credits up or down within a field's box, depending on the value in the variable `sliderVal`:

```
global sliderVal  
  
on prepareFrame  
    newVal = sliderVal * 100  
    member("Credits").scrollTop = newVal  
end
```

The global variable `sliderVal` could measure how far the user drags a slider. The statement `set newVal = sliderVal * 100` multiplies `sliderVal` to give a value that is greater than the distance the user drags the slider. If `sliderVal` is positive, the text moves up; if `sliderVal` is negative, the text moves down.

## Example

This repeat loop makes the field Credits scroll by continuously increasing the value of `scrollTop`:

```
on wa  
    member("Credits").scrollTop = 1  
    repeat with count = 1 to 150  
        member("Credits").scrollTop = member("Credits").scrollTop + 1  
        updateStage  
    end repeat  
end
```

## sds (modifier)

### Syntax

```
member(whichCastmember).model(whichModel).sds.whichProperty
```

### Description

3D modifier; adds geometric detail to models and synthesizes additional details to smooth out curves as the model moves closer to the camera. After you have added the `sds` modifier to a model using `addModifier()`, you can set the properties of the `sds` modifier.

The `sds` modifier directly affects the model resource. Be careful when using the `sds` and `lod` modifiers together, because they perform opposite functions (the `sds` modifier adds geometric detail and the `lod` modifier removes geometric detail). Before adding the `sds` modifier, it is recommended that you set the `lod.auto` modifier property to `FALSE` and set the `lod.level` modifier property to the desired resolution, as follows:

```
member("myMember").model("myModel").lod.auto = 0
member("myMember").model("myModel").lod.level = 100
member("myMember").model("myModel").addmodifier(#sds)
```

The `sds` modifier cannot be used with models that already use either the `inker` or `toon` modifiers.

After you have added the `sds` modifier to a model resource you can get or set the following properties:

`enabled` indicates whether subdivision surfaces functionality is enabled (`TRUE`) or disabled (`FALSE`). The default setting for this property is `TRUE`.

`depth` specifies the maximum number of levels of resolution that the model can display when using the `sds` modifier.

`error` indicates the level of error tolerance for the subdivision surfaces functionality. This property applies only when the `sds.subdivision` property is set to `#adaptive`.

`subdivision` indicates the mode of operation of the subdivision surfaces modifier. Possible values are as follows:

- `#uniform` specifies that the mesh is uniformly scaled up in detail, with each face subdivided the same number of times.
- `#adaptive` specifies that additional detail is added only when there are major face orientation changes and only to those areas of the mesh that are currently visible.

**Note:** For more detailed information about these properties, see the individual property entries.

### Example

The statement displays the `sds.depth` property value for the model named `Terrain`:

```
put member("3D").model("Terrain").sds.depth
-- 2
```

### See also

`lod (modifier)`, `toon (modifier)`, `inker (modifier)`, `depth (3D)`, `enabled (sds)`, `error`, `subdivision`, `addModifier`



## searchCurrentFolder

### Syntax

`the searchCurrentFolder`

### Description

System property; determines whether Director searches the current folder when searching file names. This property is `TRUE` by default.

- When the `searchCurrentFolder` property is `TRUE` (1), Director searches the current folder when resolving filenames.
- When the `searchCurrentFolder` property is `FALSE` (0), Director does not search the current folder when resolving filenames.

This property can be tested and set.

### Examples

This statement displays the status of the `searchCurrentFolder` property in the Message window:

```
put the searchCurrentFolder
```

The result is 1, which is the numeric equivalent of `TRUE`.

This statement sets the `searchCurrentFolder` property to `TRUE`, which causes Director to search the current folder when resolving filenames:

```
the searchCurrentFolder = TRUE
```

### See also

`searchPaths`

## searchPath

This is obsolete. Use `searchPaths`.

## searchPaths

### Syntax

`the searchPaths`

### Description

System property; a list of paths that Director searches when trying to find linked media such as digital video, GIFs, bitmaps, or sound files. Each item in the list is a fully qualified pathname as it appears on the current platform at run time.

The value of `searchPaths` is a linear list that you can manipulate the same as any other list by using commands such as `add`, `addAt`, `append`, `deleteAt`, and `setAt`.

URLs should not be used as file references in the search paths.

Adding a large number of paths to `searchPaths` slows searching. Try to minimize the number of paths in the list.

This property can be tested and set, and is an empty list by default.

**Note:** This property will function on all subsequent movies after being set. Because the current movie's assets have already been loaded, changing the setting will not affect any of these assets.

## Examples

This statement displays the paths that Director searches when resolving filenames:

```
put the searchPaths
```

The following statement assigns two folders to `searchPaths` in Windows. This version includes optional trailing backslashes.

```
set the searchPaths = ["c:\director\projects\", "d:\cdrom\sources\"]
```

This statement is the same, except that trailing backslashes have been omitted:

```
set the searchPaths = ["c:\director\projects", "d:\cdrom\sources"]
```

The following statement assigns two folders to `searchPaths` on a Macintosh. This version includes optional trailing colons.

```
set the searchPaths = ["hard drive:director:projects:", "cdrom:sources:"]
```

This statement is the same, except that trailing colons have been omitted:

```
set the searchPaths = ["hard drive:director:projects", "cdrom:sources"]
```

These statements cause Director to search in a folder named `Sounds`, which is in the same folder as the current Director movie:

```
set soundPaths = the moviePath & "Sounds"  
add the searchPaths, soundPath
```

## See also

`searchCurrentFolder,@ (pathname)`

# seconds

## Syntax

*dateObject.seconds*

## Description

Property; gives the seconds passed since midnight of the given date object. Only the `systemDate`, `creationDate`, and `modifiedDate` have a default `seconds` value. You must specify a `seconds` value for date objects that you create.

This property can be used with the `creationDate` and the `modifiedDate` for source control purposes.

## Example

These statements display the seconds since midnight on the authoring computer:

```
mydate = the systemdate  
put mydate.seconds  
-- 1233
```

## seek

### Syntax

```
sprite(whichSprite).seek(milliseconds)  
member(whichCastmember).seek(milliseconds)
```

### Description

RealMedia sprite or cast member method; changes the media stream's playback location to the location specified by the number of milliseconds from the beginning of the stream. The `mediaStatus` value usually becomes `#seeking` and then `#buffering`.

You can use this method to initiate play at points other than the beginning of the RealMedia stream, or to jump forward or backward in the stream. The integer specified in *milliseconds* is the number of milliseconds from the beginning of the stream; thus, to jump backward, you would specify a lower number of milliseconds, not a negative number.

If the seek command is called when `mediaStatus` is `#paused`, the stream rebuffers and returns to `#paused` at the new location specified by seek. If seek is called when `mediaStatus` is `#playing`, the stream rebuffers and automatically begins playing at the new location in the stream. If seek is called when `mediaStatus` is `#closed`, nothing happens.

If you attempt to seek beyond the stream's duration (`RealMedia`) value, the integer argument you specify is clipped to the range from 0 to the duration of the stream. You cannot jump ahead into a RealMedia sprite that is streaming live content.

Note that `x.seek(n)` is the same as `x.currentTime (RealMedia) = n`, and either of these calls will cause the stream to be rebuffered.

### Examples

The following examples set the current playback position of the stream to 10,000 milliseconds (10 seconds):

```
sprite(2).seek(10000)  
member("Real").seek(10000)
```

### See also

```
duration (RealMedia), currentTime (RealMedia), play (RealMedia), pause  
(RealMedia), stop (RealMedia), mediaStatus
```

## selectedText

### Syntax

```
member(whichTextMember).selectedText
```

### Description

Text cast member property; returns the currently selected chunk of text as a single object reference. This allows access to font characteristics as well as to the string information of the actual characters.

### Example

The following handler displays the currently selected text being placed in a local variable object. Then that object is used to reference various characteristics of the text, which are detailed in the Message window.

```
property spriteNum

on mouseUp me
    mySelectionObject = sprite(spriteNum).member.selectedText
    put mySelectionObject.text
    put mySelectionObject.font
    put mySelectionObject.fontSize
    put mySelectionObject.fontStyle
end
```

## selection() (function)

### Syntax

the selection

### Description

Function; returns a string containing the highlighted portion of the current editable field. This function is useful for testing what a user has selected in a field.

The selection function only indicates which string of characters is selected; you cannot use selection to select a string of characters.

### Example

This statement checks whether any characters are selected and, if none are, displays the alert "Please select a word.":

```
if the selection = EMPTY then alert "Please select a word."
```

### See also

selStart, selEnd

## selection (cast property)

### Syntax

```
castLib (whichCast).selection
the selection of castLib whichCast
set the selection of castLib whichCast = [ [ startMember1 , endMember1 ], \
[ startMember2 , endMember2 ], [ startMember3 , endMember3 ]... ]
castLib(whichCast). selection = [ [ startMember1 , endMember1 ], \
[ startMember2 , endMember2 ], [ startMember3 , endMember3 ]... ]
```

### Description

Cast property; determines which cast members are selected in the specified Cast window. The range appears as a list of the starting and ending cast member numbers for the selected range. You can include more than one selection by specifying additional ranges of cast members. (To specify more than one selection, Control-drag (Windows) or Command-drag (Macintosh).

This property can be tested and set.

**Example**

This statement selects cast members 1 through 10 in castLib 1:

```
castLib(1).selection = [[1, 10]]
```

This statement selects cast members 1 through 10, and 30 through 40, in castLib 1:

```
castLib(1).selection = [[1, 10], [30, 40]]
```

## selection (text cast member property)

**Syntax**

```
member(whichTextMember).selection
```

**Description**

Text cast member property; returns a list of the first and last character selected in the text cast member.

This property can be tested and set.

**Example**

The following statement sets the selection displayed by the sprite of text member myAnswer so that characters 6 through 10 are highlighted:

```
member("myAnswer").selection = [6, 10]
```

**See also**

color(), selStart, selEnd

## selEnd

**Syntax**

```
the selEnd
```

**Description**

Global property; specifies the last character of a selection. It is used with selStart to identify a selection in the current editable field, counting from the beginning character.

This property can be tested and set. The default value is 0.

**Examples**

These statements select “cde” from the field “abcdefg”:

```
the selStart = 3  
the selEnd = 5
```

This statement calls the handler noSelection when selEnd is the same as selStart:

```
if the selEnd = the selStart then noSelection
```

This statement makes a selection 20 characters long:

```
the selEnd = the selStart + 20
```

**See also**

editable, hilite (command), selection() (function), selStart, text

## selStart

### Syntax

the selStart

### Description

Cast member property; specifies the starting character of a selection. It is used with selEnd to identify a selection in the current editable field, counting from the beginning character.

This property can be tested and set. The default value is 0.

### Examples

These statements select “cde” from the field “abcdefg”:

```
the selStart = 3
the selEnd = 5
```

This statement calls the handler noSelection when selEnd is the same as selStart:

```
if the selEnd = the selStart then noSelection
```

This statement makes a selection 20 characters long:

```
the selEnd = the selStart + 20
```

### See also

selection() (function), selEnd, text

## sendAllSprites

### Syntax

sendAllSprites (*#customEvent*, *args*)

### Description

Command; sends a designated message to all sprites, not just the sprite that was involved in the event. As with any other message, the message is sent to every script attached to the sprite, unless the stopEvent command is used.

For best results, send the message only to those sprites that will properly handle the message through the sendSprite command. No error will occur if the message is sent to all the sprites, but performance may decrease. There may also be problems if different sprites have the same handler in a behavior, so avoid conflicts by using unique names for messages that will be broadcast.

After the message has been passed to all behaviors, the event follows the regular message hierarchy: cast member scripts, frame script, and then movie script.

When you use the sendAllSprites command, be sure to do the following:

- Replace *customEvent* with the message.
- Replace *args* with any arguments to be sent with the message.

If no sprite has an attached behavior containing the given handler, sendAllSprites returns FALSE.

### Example

This handler sends the custom message `allSpritesShouldBumpCounter` and the argument 2 to all sprites when the user clicks the mouse:

```
on mouseDown me
    sendAllSprites (#allSpritesShouldBumpCounter, 2)
end
```

### See also

`sendSprite`

## sendEvent

### Syntax

`member(whichCastmember).sendEvent(#eventName, arg1, arg2,...)`

### Description

3D command; sends the event *eventName*, and an arbitrary number of arguments (*arg1*, *arg2*, ...), to all scripts registered to receive the event. Use `registerForEvent()`, or `setCollisionCallback()` to register scripts for events.

### Example

The first line in this example creates an instance of a parent script named "tester". The second line sets the handler of the script instance, `jumpPluto`, as the handler to be called when the `#jump` event is sent. The third line registers a movie script handler named `jumpMars` as another handler to be called when the `#jump` event is sent. The fourth line sends the `#jump` event. The handlers `#jumpMars` in a movie script and `#jumpPluto` are called, along with any other handlers registered for the `#jump` event. Note that a script instance value of 0 indicates that you are registering a handler of a movie script, as opposed to a handler of a behavior instance or of a child of a parent script.

```
t = new (script "tester")
member("scene").registerForEvent(#jump, #jumpPluto, t)
member("scene").registerForEvent(#jump, #jumpMars, 0)
member("scene").sendEvent(#jump)
```

### See also

`registerScript()`, `registerForEvent()`, `setCollisionCallback()`

## sendSprite

### Syntax

`sendSprite (whichSprite, #customMessage, args)`

### Description

Command; sends a message to all scripts attached to a specified sprite.

Messages sent using `sendSprite` are sent to each of the scripts attached to the sprite. The messages then follow the regular message hierarchy: cast member script, frame script, and movie script.

If the given sprite does not have an attached behavior containing the given handler, `sendSprite` returns `FALSE`.

### Example

This handler sends the custom message `bumpCounter` and the argument 2 to sprite 1 when the user clicks:

```
on mouseDown me
    sendSprite (1, #bumpCounter, 2)
end
```

### See also

`sendAllSprites`

## on sendXML

### Syntax

```
sendXML "sendxmlstring", "window", "postdata"
```

### Description

Event handler; functions much like the `getURL` scripting method, which is also available using the Flash Asset Xtra. The `on sendXML` handler is called in Lingo when the `XMLObject.send` `ActionScript` method is executed in a Flash sprite or Flash XML object.

In `ActionScript`, the `XMLObject.send` method passes two parameters in addition to the XML data in the XML object. These parameters are as follows:

- *url*—the URL to send the XML data to. Usually this is the URL of a server script that is waiting to process the XML data.
- *window*—the browser window in which to display the server's response data.

The `ActionScript XMLObject.send` method can be called in Director either by a Flash sprite or by a global Flash XML object created in Lingo. When this happens, the Lingo `on sendXML` handler is called, and the same parameters are passed to the handler.

The following Lingo illustrates how the parameters are received by the `on sendXML` handler:

```
on sendXML me, theURL, targetWindow, XMLdata
```

These parameters correlate with the `XMLObject.send` parameters as follows:

- *theURL*—the URL to send the XML data to.
- *targetWindow*—the browser window in which to display the server's response.
- *XMLdata*—the XML data in the Flash XML object.

By creating an `on sendXML` handler in your Director movie, you enable it to process `XMLObject.send` events generated in a Flash sprite or a global Flash object.

Flash sprites can also load external XML data or parse internal XML data. The Flash Asset Xtra handles these functions in the same way as a Flash 5 or Flash MX movie in your browser.

### Example

This Lingo command gets the `XMLObject.send` method information from a Flash sprite and then directs the browser to the URL and transmits the XML data to the URL:

```
on sendXML me, theURL, targetWindow, xmlData
    gotoNetPage theURL, targetWindow
    postNetText(theURL, xmlData)
end
```



# serialNumber

## Syntax

the serialNumber

## Description

Movie property; a string containing the serial number entered when Director was installed.

This property is available in the authoring environment only. It could be used in a movie in a window (MIAW) tool that is personalized to show the user's information.

## Example

This handler would be located in a movie script of a MIAW. It places the user's name and the serial number into a display field when the window is opened:

```
on prepareMovie
    displayString = the userName
    put RETURN&the organizationName after displayString
    put RETURN&the serialNumber after displayString
    member("User Info").text = displayString
end
```

## See also

organizationName, userName, window

# set...to, set...=

## Syntax

```
set the lingoProperty to expression
the lingoProperty = expression
set variable to expression
variable = expression
```

## Description

Command; evaluates an expression and puts the result in the property specified by *lingoProperty* or the variable specified by *variable*.

## Examples

This statement sets the name of member 3 to Sunset:

```
set member(3).name = "Sunset"
```

The following statement sets the soundEnabled property to the opposite of its current state. When soundEnabled is TRUE (the sound is on), this statement turns it off. When soundEnabled is FALSE (the sound is off), this statement turns it on.

```
set the soundEnabled = not (the soundEnabled)
```

This statement sets the variable vowels to the string "aeiou":

```
set vowels = "aeiou"
```

## See also

property

# setAlpha()

## Syntax

```
imageObject.setAlpha(alphaLevel)
imageObject.setAlpha(alphaImageObject)
```

## Description

Function; sets the alpha channel of an image object to a flat *alphaLevel* or to an existing *alphaImageObject*. The *alphaLevel* must be a number from 0–255. Lower values cause the image to appear more transparent. Higher values cause the image to appear more opaque. The value 255 has the same effect as a value of zero. In order for the *alphaLevel* to have effect, the *useAlpha()* of the image object must be set to TRUE.

The image object must be 32-bit. If you specify an alpha image object, it must be 8-bit. Both images must have the same dimensions. If these conditions are not met, *setAlpha()* has no effect and returns FALSE. The function returns TRUE when it is successful.

## Examples

The following Lingo statement makes the image of the bitmap cast member Foreground opaque and disables the alpha channel altogether. This is a good method for removing the alpha layer from an image:

```
member("Foreground").image.setAlpha(255)
member("Foreground").image.useAlpha = FALSE
```

This Lingo gets the alpha layer from the cast member Sunrise and places it into the alpha layer of the cast member Sunset:

```
tempAlpha = member("Sunrise").image.extractAlpha()
member("Sunset").image.setAlpha(tempAlpha)
```

## See also

*useAlpha()*, *extractAlpha()*

# setaProp

## Syntax

```
setaProp list, listProperty, newValue
setaProp (childObject, listProperty, newValue)
list.listProperty = newValue
list[listProperty] = newValue
childObject.listProperty = newValue
```

## Description

Command; replaces the value assigned to *listProperty* with the value specified by *newValue*. The *setaProp* command works with property lists and child objects. Using *setaProp* with a linear list produces a script error.

- For property lists, *setaProp* replaces a property in the list specified by *list*. When the property isn't already in the list, Lingo adds the new property and value.
- For child objects, *setaProp* replaces a property of the child object. When the property isn't already in the object, Lingo adds the new property and value.
- The *setaProp* command can also set ancestor properties.

## Examples

These statements create a property list and then adds the item `#c:10` to the list:

```
newList = [#a:1, #b:5]
put newList
-- [#a:1, #b:5]
setaProp newList, #c, 10
put newList
```

Using the dot operator, you can alter the property value of a property already in a list without using `setaProp`:

```
newList = [#a:1, #b:5]
put newList
-- [#a:1, #b:5]
newList.b = 99
put newList
-- [#a:1, #b:99]
```

**Note:** To use the dot operator to manipulate a property, the property must already exist in the list, child object, or behavior.

## See also

ancestor, property, . (dot operator)

# setAt

## Syntax

```
setAt list, orderNumber, value
list[orderNumber] = value
```

## Description

Command; replaces the item specified by *orderNumber* with the value specified by *value* in the list specified by *list*. When *orderNumber* is greater than the number of items in a property list, the `setAt` command returns a script error. When *orderNumber* is greater than the number of items in a linear list, Director expands the list's blank entries to provide the number of places specified by *orderNumber*.

## Examples

This handler assigns a name to the list `[12, 34, 6, 7, 45]`, replaces the fourth item in the list with the value 10, and then displays the result in the Message window:

```
on enterFrame
    set vNumbers = [12, 34, 6, 7, 45]
    setAt vNumbers, 4, 10
    put vNumbers
end enterFrame
```

When the handler runs, the Message window displays the following:

```
[12, 34, 6, 10, 45]
```

You can perform this same operation may be done using bracket access to the list in the following manner:

```
on enterFrame
    set vNumbers = [12, 34, 6, 7, 45]
    vNumbers[4] = 10
    put vNumbers
end enterFrame
```

When the handler runs, the Message window displays the following:

```
[12, 34, 6, 10, 45]
```

**See also**

[ ] (bracket access)

## setCallback()

**Syntax**

```
flashSpriteReference.setCallback(actionScriptObject,ASEventName,#LingoHandlerName,  
                                lingoScriptObject)  
setCallback(actionScriptObject,ASEventName,#LingoHandlerName,lingoScriptObject)
```

**Description**

Flash command; this command can be used as a sprite or a global method to define a Lingo callback handler for a particular event generated by the specified object. When ActionScript triggers the event in the object, that event is redirected to the given Lingo handler, including all arguments that are passed with the event.

If the ActionScript object was originally created within a Flash sprite, use the *flashSpriteReference* syntax. If the object was originally created globally, use the global syntax.

**Note:** If you have not imported any Flash cast members, you must manually add the Flash Asset Xtra to your movie's Xtra list in order for global Flash commands to work correctly. You add Xtra extensions to the Xtra list by choosing **Modify > Movie > Xtras**. For more information, see *Managing Xtra extensions for distributed movies in Using Director*.

**Examples**

This statement sets a the Lingo handler named `myOnStatus` in the Lingo script object `me` to be called when an `onStatus` event is generated by the ActionScript object `tLocalConObject` in the Flash movie in sprite 3:

```
sprite(3).setCallback(tLocalConObject, "onStatus", #myOnStatus, me)
```

This statement sets the Lingo handler named `myOnStatus` in the Lingo script object `me` to be called when an `onStatus` event is generated by the global ActionScript object `tLocalConObject`:

```
setCallback(tLocalConObject, "onStatus", #myOnStatus, me)
```

The following statements create a new global XML object and create a callback handler that parses XML data when it arrives. The third line loads an XML file. The callback handler is included as well.

```
gXMLCB = newObject("XML")  
setCallback( gXMLCB, "onData", #dataFound, 0 )  
gXMLCB.load( "myfile.xml" )  
  
-- Callback handler invoked when xml data arrives  
on dataFound me, obj, source  
    obj.parseXML(source)  
    obj.loaded = 1  
    obj.onload(TRUE)  
end dataFound
```

**See also**

```
newObject(), clearAsObjects()
```

## setCollisionCallback()

### Syntax

```
member(whichCastmember).model(whichModel).collision.\  
    setCollisionCallback (#handlerName, scriptInstance)
```

### Description

3D collision command; registers the handler *#handlerName* in the given *scriptInstance* to be called when *whichModel* is involved in a collision.

This command works only if the model's `collision.enabled` property is `TRUE`. The default behavior is determined by the value of `collision.resolve`, you can override it using the `collision.resolveA` and/or the `collision.resolveB` commands. Do not use the `updateStage` command in the specified handler.

This command is a shorter alternative to using the `registerScript` command for collisions, but there is no difference in the overall result. This command can be considered to perform a small subset of the `registerScript` command functionality.

### Example

This statement causes the *#bounce* handler in the cast member *colScript* to be called when the model named *Sphere* collides with another model:

```
member("3d world").model("Sphere").collision.\  
    setCollisionCallback\  
(#bounce, member("colScript"))
```

### See also

`collisionData`, `collision (modifier)`, `resolve`, `resolveA`, `resolveB`, `registerForEvent()`, `registerScript()`, `sendEvent`

## setFlashProperty()

### Syntax

```
sprite(spriteNum).setFlashProperty(targetName, #property, newValue)
```

### Description

Function; allows Lingo to call the Flash action script function `setProperty()` on the given Flash sprite. Use the `setFlashProperty()` function to set the properties of movie clips or levels within a Flash movie. This is similar to setting sprite properties within Director.

The *targetName* is the name of the movie clip or level whose property you want to set within the given Flash sprite.

The *#property* is the name of the property to set. You can set the following movie clip properties: *#posX*, *#posY*, *#scaleX*, *#scaleY*, *#visible*, *#rotate*, *#alpha*, and *#name*.

To set a global property of the Flash sprite, pass an empty string as the *targetName*. You can set the global Flash properties: *#focusRect* and *#spriteSoundBufferTime*.

See the Flash documentation for descriptions of these properties.

### Example

This statement sets the value of the `#rotate` property of the movie clip `Star` in the Flash member in sprite 3 to 180:

```
sprite(3).setFlashProperty("Star", #rotate, 180)
```

### See also

`getFlashProperty()`

## setPixel()

### Syntax

```
imageObject.setPixel(x, y, colorObject)
imageObject.setPixel(point(x, y), colorObject)
imageObject.setPixel(x, y, integerValue)
imageObject.setPixel(point(x, y), integerValue)
```

### Description

Function; sets the color value of the pixel at the specified point in the given image object, to either *colorObject* or to a raw integer with *integerValue*. If you are setting many pixels to the color of another pixel with `getPixel()`, it is faster to set them as raw integers.

For best performance with color objects, with 8-bit or lower images use an indexed color object, and with 16-bit or higher images, use an RGB color object.

The `SetPixel()` function returns `FALSE` if the specified pixel falls outside the specified image.

To see an example of `setPixel()` used in a completed movie, see the Imaging movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This Lingo statement draws a horizontal black line 50 pixels from left to right in cast member 5:

```
repeat with x = 1 to 50
    member(5).image.setPixel(x, 0, rgb(0, 0, 0))
end repeat
```

### See also

`getPixel()`, `draw()`, `fill()`, `color()`

## setPlaylist()

### Syntax

```
sound(channelNum).setPlaylist([#membermember(whichmember){#startTime:milliseconds,
    #endTime:milliseconds,#loopCount:numberOfLoops,#loopStartTime:milliseconds,
    #loopEndTime: milliseconds, #preloadTime: milliseconds}]. . . )
setPlaylist(sound(channelNum)[#membermember(whichmember){#startTime:milliseconds,
    #endTime:milliseconds,#loopCount:numberOfLoops,#loopStartTime:milliseconds,
    #loopEndTime: milliseconds, #preloadTime: milliseconds}]. . . )
```

### Description

Function; sets or resets the playlist of the given sound channel. This command is useful for queueing several sounds at once.

You can specify these parameters for each sound to be queued:

Property	Description
#member	The sound cast member to queue. This property must be provided; all others are optional.
#startTime	The time within the sound at which playback begins, in milliseconds. The default is the beginning of the sound. See <code>startTime</code> .
#endTime	The time within the sound at which playback ends, in milliseconds. The default is the end of the sound. See <code>endTime</code> .
#loopCount	The number of times to play a loop defined with <code>#loopStartTime</code> and <code>#loopEndTime</code> . The default is 1. See <code>loopCount</code> .
#loopStartTime	The time within the sound to begin a loop, in milliseconds. See <code>loopStartTime</code> .
#loopEndTime	The time within the sound to end a loop, in milliseconds. See <code>loopEndTime</code> .
#preloadTime	The amount of the sound to buffer before playback, in milliseconds. See <code>preloadTime</code> .

To see an example of `setPlaylist()` used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This handler queues and plays the cast member `introMusic`, starting at its 3-second point, with a loop repeated 5 times from the 8-second point to the 8.9-second point, and stopping at the 10-second point.

```
on playMusic
  sound(2).queue([#member: member("introMusic"), #startTime: 3000,\
    #endTime: 10000, #loopCount: 5, #loopStartTime: 8000, #loopEndTime: 8900])
  sound(2).play()
end
```

### See also

`endTime`, `getPlaylist()`, `startTime`, `loopCount`, `loopEndTime`, `loopStartTime`, `member` (sound property), `play()` (sound), `preloadTime`, `queue()`

## setPref

### Syntax

```
setPref prefName, prefValue
```

### Description

Command; writes the string specified by *prefValue* in the file specified by *prefName* on the computer's local disk. The file is a standard text file.

The *prefName* argument must be a valid filename. To make sure the filename is valid on all platforms, use no more than eight alphanumeric characters for the file name.

After the `setPref` command runs, if the movie is playing in a browser, a folder named `Prefs` is created in the Plug-In Support folder. The `setPref` command can write only to that folder.

If the movie is playing in a projector or Director, a folder is created in the same folder as the application. The folder receives the name *Prefs*.

Do not use this command to write to read-only media. Depending on the platform and version of the operating system, you may encounter errors or other problems.

This command does not perform any sophisticated manipulation of the string data or its formatting. You must perform any formatting or other manipulation in conjunction with `getPref()`; you can manipulate the data in memory and write it over the old file using `setPref`.

In a browser, data written by `setPref` is not private; any Shockwave movie can read this information and upload it to a server. Do not store confidential information using `setPref`.

On Windows, the `setPref` command fails if the user is a restricted user.

To see an example of `setPref` used in a completed movie, see the Read and Write Text movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This handler saves the contents of the field cast member Text Entry in a file named DayWare settings:

```
on mouseUp me
    setPref "CurPrefs", member("Text Entry").text
end
```

### See also

`getPref()`

## setProp

### Syntax

```
setProp list, property, newValue
list.listProperty = newValue
list[listProperty] = newValue
```

### Description

Command; replaces the value assigned to *property* with the value specified by *newValue* in the list specified by *list*. If the list doesn't contain the specified property, `setProp` returns a script error.

The `setProp` command works with property lists only. Using `setProp` with a linear list produces a script error.

This command is similar to the `setaProp` command, except that `setProp` returns an error when the property is not already in the list.

### Examples

This statement changes the value assigned to the age property of property list x to 11:

```
setProp x, #age, 11
```

Using the dot operator, you can alter the property value of a property already in a list, exactly as above:

```
x.age = 11
```

### See also

`setaProp`



## setScriptList()

### Syntax

```
spriteReference.setScriptList(scriptList)  
sprite(whichSprite).setScriptList(scriptList)
```

### Description

This command sets the `scriptList` of the given sprite. The `scriptList` indicates which scripts are attached to the sprite and what the settings of each script property are. By setting this list, you can change which behaviors are attached to a sprite or change the behavior properties.

The list takes the form:

```
[ [ (whichBehaviorMember), " [ #property1: value, #property2: value, . . . ] " ,  
[ (whichBehaviorMember), " [ #property1: value, #property2: value, . . . ] " ] ]
```

This command cannot be used during a score recording session. Use `setScriptList()` for sprites added during score recording after the score recording session has ended.

### See also

`scriptList`, `value()`, `string()`

## settingsPanel()

**Help ID:** x5540 | Lingo\_FlashSettingsPanel

### Syntax

```
spriteReference.settingsPanel({integerPanelIndex})  
sprite(whichSprite).settingsPanel({integerPanelIndex})
```

### Description

Flash sprite command; invokes the Flash Settings dialog box to the specified panel index. This is the same dialog box that can be opened by right-clicking (Windows) or Control-clicking (Macintosh) on a Flash movie playing in a browser. The `integerPanelIndex` can have a value of 0, 1, 2, or 3, indicating which panel to activate when the dialog box is opened. A value of 0 opens the dialog box showing the Privacy tab, a value of 1 opens it showing the Local Storage tab, a value of 2 opens it showing the Microphone tab, and a value of 3 opens it showing the Camera tab. The default panel index is 0.

The Settings dialog box will not be displayed if the Flash sprite's rectangle is not large enough to accommodate it.

If you want to emulate the Flash Player by invoking the Settings dialog box when a user right-clicks (Windows) or Control-clicks (Macintosh), you can use this command in a `mouseDown` handler that tests for the `rightMouseDown` property or the `controlDown` property.

In order to emulate the Flash Player by enabling the Settings dialog box in a Director movie running in a browser, you must first disable the Shockwave context menu that is available by right-clicking (Windows) or Control-clicking (Macintosh) on a Shockwave movie playing in a browser. For information on how to disable this menu, see "Using the Flash Settings panel" in Director Help (Help > Using Director).

### Example

This statement opens the Flash Settings panel with the Local Storage tab active:

```
sprite(3).settingsPanel(1)
```

### See also

on mouseDown (event handler), rightMouseDown (system property), controlDown

## setTrackEnabled

### Syntax

```
sprite(whichSprite).setTrackEnabled(whichTrack, trueOrFalse )  
setTrackEnabled(sprite whichSprite, whichTrack, trueOrFalse)
```

### Description

Command; determines whether the specified track in the digital video is enabled to play.

- When `setTrackEnabled` is TRUE, the specified track is enabled and playing.
- When `setTrackEnabled` is FALSE, the specified track is disabled and muted. For video tracks, this means they will no longer be updated on the screen.

To test whether a track is already enabled, test the `trackEnabled` sprite property.

### Example

This statement enables track 3 of the digital video assigned to sprite channel 8:

```
sprite(8).setTrackEnabled(3, TRUE)
```

### See also

`trackEnabled`

## setVariable()

### Syntax

```
setVariable(sprite flashSpriteNum, "variableName", newValue)
```

### Description

Function; sets the value of the given variable in the given Flash sprite. Flash variables were introduced in Flash version 4.

### Example

The following statement sets the value of the variable `currentURL` in the Flash cast member in sprite 3. The new value of `currentURL` will be “`http://www.macromedia.com/software/flash/`”.

```
setVariable(sprite 3, "currentURL", "http://www.macromedia.com/software/flash/")
```

### See also

`hitTest()`, `getVariable()`

# shader

## Syntax

```
member(whichCastmember).shader(whichShader)  
member(whichCastmember).shader[index]  
member(whichCastmember).model(whichModel).shader  
member(whichCastmember).modelResource(whichModelResource).\  
    face[index].shader
```

## Description

3D element, model property, and face property; the object used to define the appearance of the surface of the model. The shader is the “skin” which is wrapped around the model resource used by the model.

The shader itself is not an image. The visible component of a shader is created with up to eight layers of texture. These eight texture layers are either created from bitmap cast members or image objects within Director or imported with models from 3D modeling programs. For more information, see [texture](#).

Every model has a linear list of shaders called the `shaderList`. The number of entries in this list equals the number of meshes in the model resource used by the model. Each mesh can be shaded by only one shader.

The 3D cast member has a default shader named `DefaultShader`, which cannot be deleted. This shader is used when no shader has been assigned to a model and when a shader being used by a model is deleted.

The syntax `member(whichCastmember).model(whichModel).shader` gives access to the first shader in the model's `shaderList` and is equivalent to

```
member(whichCastmember).model(whichModel).shaderList[1].
```

Create and delete shaders with the `newShader()` and `deleteShader()` commands.

Shaders are stored in the shader palette of the 3D cast member. They can be referenced by name (*whichShader*) or palette index (*shaderIndex*). A shader can be used by any number of models. Changes to a shader will appear in all models which use that shader.

There are four types of shaders:

`#standard` shaders present their textures realistically.

`#painter`, `#engraver`, and `#newsprint` shaders stylize their textures for painting, engraving, and newsprint effects. They have special properties in addition to the `#standard` shader properties.

For a complete list of shader properties, see [3D Lingo by Feature](#).

The shaders used by individual faces of `#mesh` primitives can be set with the syntax

```
member(whichCastmember).modelResource(whichModelResource).  
face[index].shader. Changes to this property require a call to the build() command.
```

## Example

This statement sets the shader property of the model named `Wall` to the shader named `WallSurface`:

```
member("Room").model("Wall").shader = \  
    member("Room").shader("WallSurface")
```

## See also

`shaderList`, `newShader`, `deleteShader`, `face`, `texture`

# shaderList

## Syntax

```
member(whichCastmember).model(whichModel).shaderList  
member(whichCastmember).model(whichModel).shaderList[index]
```

## Description

3D model property; a linear list of `shadowPercentage` applied to the model. The number of entries in this list equals the number of meshes in the model resource used by the model. Each mesh can be shaded by only one shader.

Set the shader at the specified *index* position in the `shaderList` with this syntax:

```
member(whichCastmember).model(whichModel).shaderList[index] = shaderReference
```

With 3D text, each character is a separate mesh. Set the value of *index* to the number of the character whose shader you want to set.

Set all *index* positions in the `shaderList` to the same shader with this syntax (note the absence of an index for the `shaderList`):

```
member(whichCastmember).model(whichModel).shaderList  
= \
```

```
shaderReference
```

Set a property of a shader in the `shaderList` with this syntax:

```
member(whichCastmember).model(whichModel).shaderList[index].\  
whichProperty = propValue
```

Set a property of all of the shaders of a model to the same value with this syntax (note the absence of an index for the `shaderList`):

```
member(whichCastmember).model(whichModel).shaderList.\  
whichProperty = propValue
```

## Examples

This statement sets the second shader in the `shaderList` of the model named `Bumper` to the shader named `Chrome`:

```
member("Car").model("Bumper").shaderList[2] = \  
    member("Car").shader("Chrome")
```

This statement sets the all the shaders in the `shaderList` of the model named `Bumper` to the shader named `Chrome`:

```
member("Car").model("Bumper").shaderList = \  
    member("Car").shader("Chrome")
```

## See also

`shadowPercentage`

# shadowPercentage

## Syntax

```
member(whichCastmember).model(whichModel).toon.shadowPercentage  
member(whichCastmember).model(whichModel).shader.shadowPercentage  
member(whichCastmember).shader(whichShader).shadowPercentage
```

## Description

3D toon modifier and painter shader property; indicates the percentage of available colors that are used in the area of the model's surface where light does not create highlights.

The range of this property is 0 to 100, and the default value is 50.

The number of colors used by the `toon` modifier and `painter` shader for a model is determined by the `colorSteps` property of the model's `toon` modifier or `painter` shader.

#### Example

The following statement sets the `shadowPercentage` property of the `toon` modifier for the model named `Teapot` to 50. Half of the colors available to the `toon` modifier for this model will be used for the shadow area of the model's surface.

```
member("shapes").model("Teapot").toon.shadowPercentage = 50
```

#### See also

`colorSteps`, `shadowStrength`

## shadowStrength

#### Syntax

```
member(whichCastmember).model(whichModel).toon.shadowStrength  
member(whichCastmember).model(whichModel).shader.shadowStrength  
member(whichCastmember).shader(whichShader).shadowStrength
```

#### Description

3D `toon` modifier and `#painter` shader property; indicates the brightness of the area of the model's surface where light does not create highlights.

The default value of this property is 1.0.

#### Example

The following statement sets the `shadowStrength` property of the `toon` modifier for the model named `Sphere` to 0.1. The area of the model's surface that is not highlighted will be very dark.

```
member("Shapes").model("Sphere").toon.shadowStrength = 0.1
```

## shapeType

#### Syntax

```
member(whichCastMember).shapeType  
the shapeType of member whichCastMember
```

#### Description

Shape cast member property; indicates the specified shape's type. Possible types are `#rect`, `#roundRect`, `#oval`, and `#line`. You can use this property to specify a shape cast member's type after creating the shape cast member using Lingo.

#### Example

These statements create a new shape cast member numbered 100 and then define it as an oval:

```
new(#shape, member 100)  
member(100).shapeType = #oval
```

## shiftDown

### Syntax

the shiftDown

### Description

System property; indicates whether the user is pressing the Shift key (TRUE) or not (FALSE).

In the Director player for Java, this function returns TRUE only if the Shift key and another key are pressed simultaneously. If the Shift key is pressed by itself, shiftDown returns FALSE.

The Director player for Java supports key combinations with the Shift key. However, the browser receives the keys before the movie and thus responds to and intercepts any key combinations that are also browser keyboard shortcuts.

This property must be tested in conjunction with another key.

### Example

This statement checks whether the Shift key is being pressed and calls the handler doCapitalA if it is:

```
if (the shiftDown) then doCapitalA (the key)
```

### See also

commandDown, controlDown, key(), optionDown

## shininess

### Syntax

```
member(whichCastmember).shader(whichShader).shininess  
member(whichCastmember).model(whichModel).shader.shininess  
member(whichCastmember).model(whichModel).shaderList\  
[shaderListIndex].shininess
```

### Description

3D standard shader property; allows you to get or set the shininess of a surface. Shininess is defined as the percentage of shader surface devoted to highlights. The value is an integer between 0 and 100, with a default of 30.

All shaders have access to the #standard shader properties; in addition to these standard shader properties shaders of the types #engraver, #newsprint, and #painter have properties unique to their type. For more information, see newShader.

### Example

The following statement sets the shininess property of the first shader in the shader list of the model gbCyl3 to 60. Sixty percent of the surface of the shader will be dedicated to highlights.

```
member("Scene").model("gbCyl3").shader.shininess = 60
```

## short

### See

`date()` (system clock), `time()`

## showGlobals

### Syntax

```
showGlobals
```

### Description

Command; displays all global variables in the Message window. This command is useful for debugging scripts.

### Example

This statement displays all global variables in the Message window:

```
showGlobals
```

### See also

`clearGlobals`, `global`, `globals`

## showLocals

### Syntax

```
showLocals
```

### Description

Command; displays all global variables in the Message window. This command is useful only within handlers or parent scripts that contain local variables to display. All variables used in the Message window are automatically global.

Local variables in a handler are no longer available after the handler executes. Inserting the statement

```
showLocals
```

in a handler displays all the local variables in that handler in the Message window.

This command is useful for debugging scripts.

### See also

`clearGlobals`, `global`, `showGlobals`

## showProps()

### Syntax

```
member(whichFlashOrVectorCastMember). showProps  
showProps()  
sprite(whichFlashOrVectorSprite).showProps()  
sound(channelNum).showProps()
```

### Description

Command; displays a list of the current property settings of a Flash movie, Vector member, or currently playing sound in the Message window. This command is useful for authoring only; it does not work in projectors or in Shockwave movies.

### Example

This handler accepts the name of a cast as a parameter, searches that cast for Flash movie cast members, and displays the cast member name, number, and properties in the Message window:

```
on ShowCastProperties whichCast  
  repeat with i = 1 to the number of members of castLib whichCast  
    castType = member(i, whichCast).type  
    if (castType = #flash) OR (castType = #vectorShape) then  
      put castType&&"cast member" && i & ":" && member(i, whichCast).name  
      put RETURN  
      member(i, whichCast).showProps()  
    end if  
  end repeat  
end
```

### See also

queue(), setPlaylist()

## showResFile

### Description

This Lingo is obsolete.

## showXlib

### Syntax

```
showXlib {Xlibfilename}
```

### Description

Command; shows all Xtra extensions in *Xlibfilename* (which must be open), or all open Xlibraries if no file is specified. Xlibrary files are resource files that contain DLLs (Windows) or Xtra resources (Macintosh). Because the types of Xlibrary files in Windows and on the Macintosh differ, the list of files that the showXlib command generates is different on different platforms.

The showXlib command doesn't support URLs as file references.

You can use `interface()` to display online documentation for an Xtra.

**Note:** This command is not supported in Shockwave.



**Example**

This statement displays the Xtra extensions in the VideoDisc Xlibrary:

```
showXlib "VideoDisc Xlibrary"
```

**See also**

```
closeXlib, interface(), openXlib
```

## shutDown

**Syntax**

```
shutDown
```

**Description**

Command; closes all open applications and turns off the computer.

**Example**

This statement checks whether the user has pressed Control+S (Windows) or Command+S (Macintosh) and, if so, shuts down the computer:

```
if the key = "s" and the commandDown then
    shutDown
end if
```

**See also**

```
quit, restart
```

## silhouettes

**Syntax**

```
member(whichCastmember).model(whichModel).inker.silhouettes
member(whichCastmember).model(whichModel).toon.silhouettes
```

**Description**

3D `toon` and `inker` modifier property; indicates the presence (TRUE) or absence (FALSE) of lines drawn by the modifier at the visible edges of the model.

Silhouette lines are drawn around the model's 2D image on the camera's projection plane. Their relationship to the model's mesh is not fixed, unlike crease or boundary lines, which are drawn on features of the mesh.

Silhouette lines are similar to the lines that outline images in a child's coloring book.

The default value for this property is TRUE.

**Example**

The following statement sets the `silhouettes` property of the `inker` modifier for the model named Sphere to FALSE. Lines will not be drawn around the profile of the model.

```
member("Shapes").model("Sphere").inker.silhouettes = FALSE
```

## sin()

### Syntax

`sin(angle)`

### Description

Math function; calculates the sine of the specified angle. The angle must be expressed in radians as a floating-point number.

### Example

This statement calculates the sine of pi/2:

```
put sin (PI/2.0)
-- 1
```

### See also

PI

## size

### Syntax

`member(whichCastMember).size`  
the size of member *whichCastMember*

### Description

Cast member property; returns the size in memory, in bytes, of a specific cast member number or name. Divide bytes by 1024 to convert to kilobytes.

### Example

This line outputs the size of the cast member Shrine in a field named How Big:

```
member("How Big").text = string(member("shrine").size)
```

## sizeRange

### Syntax

```
member(whichCastmember).modelResource
(whichModelResource).sizeRange.start
modelResourceObjectReference.sizeRange.start
member(whichCastmember).modelResource
(whichModelResource).sizeRange.end
modelResourceObjectReference.sizeRange.end
```

### Description

3D property; when used with a model resource whose type is #particle, this property allows you to get or set the start and end property of the model resource's sizeRange. Particles are measured in world units.

The size of particles in the system is interpolated linearly between `sizeRange.start` and `sizeRange.end` over the lifetime of each particle.

This property must be an integer greater than 0, and has a default value of 1.

### Example

In this example, `mrFount` is a model resource of the type `#particle`. This statement sets the `sizeRange` properties of `mrFount`. The first line sets the start value to 4, and the second line sets the end value to 1. The effect of this statement is that the particles of `mrFount` are size 4 when they first appear, and gradually shrink to a size of 1 during their lifetime.

```
member("fountain").modelResource("mrFount").sizeRange.start = 4
member("fountain").modelResource("mrFount").sizeRange.end = 1
```

## skew

### Syntax

```
sprite(whichSpriteNumber).skew
```

### Description

Sprite property; returns, as a float value in hundredths of a degree, the angle to which the vertical edges of the sprite are tilted (skewed) from the vertical. Negative values indicate a skew to the left; positive values indicate a skew to the right. Values greater than 90° flip an image vertically.

The Score can retain information for skewing an image from +21,474,836.47° to -21,474,836.48°, allowing 59,652 full rotations in either direction.

When the skew limit is reached (slightly past the 59,652th rotation), the skew resets to +116.47° or -116.48°—not 0.00°. This is because +21,474,836.47° is equal to +116.47°, and -21,474,836.48° is equal to -116.48° (or +243.12°). To avoid this reset condition, constrain angles to ±360° in either direction when using Lingo to perform continuous skewing.

### Example

The following behavior causes a sprite to skew continuously by 2° every time the playhead advances, while limiting the angle to 360°:

```
property spriteNum

on prepareFrame me
    sprite(spriteNum).skew = integer(sprite(spriteNum).skew + 2) mod 360
end
```

### See also

`flipH`, `flipV`, `rotation`

## smoothness

### Syntax

```
member(whichTextmember).smoothness
member(whichCastMember).modelResource(whichExtruderModelResource)\
    .smoothness
```

### Description

3D extruder model resource and text property; allows you to get or set an integer controlling the number of segments used to create a 3D text cast member. The higher the number, the smoother the text appears. The range of this property is 1 to 10, and the default value is 5.

For more information about working with extruder model resources and text cast members, see `extrude3D`.

### Example

In this example, the cast member Logo is a text cast member. This statement sets the smoothness of Logo to 8. When Logo is displayed in 3D mode, the edges of its letters will be very smooth.

```
member("Logo").smoothness = 8
```

In this example, the model resource of the model Slogan is extruded text. This statement sets the smoothness of Slogan's model resource to 1, causing the Slogan's letters to appear very angular.

```
member("Scene").model("Slogan").resource.smoothness = 1
```

### See also

extrude3D

## sort

### Syntax

```
list.sort()  
sort list
```

### Description

Command; puts list items into alphanumeric order.

- When the list is a linear list, the list is sorted by values.
- When the list is a property list, the list is sorted alphabetically by properties.

After a list is sorted, it maintains its sort order even when you add new variables using the add command.

### Example

The following statement puts the list Values, which consists of [#a: 1, #d: 2, #c: 3], into alphanumeric order. The result appears below the statement.

```
put values  
-- [#a: 1, #d: 2, #c: 3]  
values.sort()  
put values  
--[#a: 1, #c: 3, #d: 2]
```

## sound

### Syntax

```
member(whichCastMember).sound  
the sound of member whichCastMember
```

### Description

Cast member property; controls whether a movie, digital video, or Flash movie's sound is enabled (TRUE, default) or disabled (FALSE). In flash members, the new setting takes effect after the currently playing sound ends.

This property can be tested and set.

To see an example of sound used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This handler accepts a member reference and toggles the member's `sound` property on or off:

```
on ToggleSound whichMember
    member(whichMember).sound = not member(whichMember).sound
end
```

## soundBusy()

### Syntax

```
soundBusy(whichChannel)
```

### Description

Function; determines whether a sound is playing (TRUE) or not playing (FALSE) in the sound channel specified by *whichChannel*.

Make sure that the playhead has moved before using `soundBusy()` to check the sound channel. If this function continues to return FALSE after a sound should be playing, add the `updateStage` command to start playing the sound before the playhead moves again.

This function works for those sound channels occupied by actual audio cast members. QuickTime, Flash, and Shockwave audio handle sound differently, and this function will not work with those media types.

Consider using the `status` property of a sound channel instead of `soundBusy()`. Status can be more accurate under many circumstances.

### Example

The following statement checks whether a sound is playing in sound channel 1 and loops in the frame if it is. This allows the sound to finish before the playhead goes to another frame.

```
if soundBusy(1) then go to the frame
```

### See also

```
sound playFile, sound stop, status
```

## soundChannel (SWA)

### Syntax

```
member(whichCastMember).soundChannel  
the soundChannel of member whichCastMember
```

### Description

Shockwave Audio (SWA) cast member property; specifies the sound channel in which the SWA sound plays.

If no channel number or channel 0 is specified, the SWA streaming cast member assigns the sound to the highest numbered sound channel that is unused.

Shockwave Audio streaming sounds can appear as sprites in sprite channels, but they play sound in a sound channel. Refer to SWA sound sprites by their sprite channel number, not their sound channel number.

This property can be tested and set.

### Example

This statement tells the SWA streaming cast member Frank Zappa to play in sound channel 3:

```
member("Frank Zappa").soundChannel = 3
```

## soundChannel (RealMedia)

### Syntax

```
sprite(whichSprite).soundChannel  
member(whichCastmember).soundChannel  
sprite(whichSprite).soundChannel = soundChannel  
member(whichCastmember).soundChannel = soundChannel
```

### Description

RealMedia sprite or cast member property; allows you to get or set the sound channel used to play the audio in the RealMedia stream. Setting this property allows you to control the audio of a RealMedia stream using the Lingo sound methods and properties. Setting this property to a value outside the range 0–8 causes a Lingo error. This property has no effect if `realPlayerNativeAudio()` is set to TRUE.

The default setting for this property is 0, which means that the RealMedia audio will play in the highest sound channel available, and the property's value changes during playback depending on which channel is being used. When the RealMedia cast member is playing, this property reflects the sound channel currently in use. When the RealMedia cast member is stopped, this property reverts to 0.

If you specify a channel (1–8) for this property and there are sounds currently playing in that channel (from other parts of the movie), they will be stopped and the RealMedia audio will play in the channel instead.

Concurrently playing RealMedia cast members are not supported; if your movie contains RealMedia cast members that play concurrently, their sounds are played simultaneously in the same sound channel,

### Examples

The following examples show that the sound in the RealMedia stream in sprite 2 and the cast member Real will be played in sound channel 2.

```
put sprite(2).soundChannel  
-- 2  
put member("Real").soundChannel  
-- 2
```

The following examples assign sound channel 1 to the RealMedia stream in sprite 2 and the cast member Real.

```
sprite(2).soundChannel = 1  
member("Real").soundChannel = 1
```

### See also

```
realPlayerNativeAudio()
```

## sound close

This is obsolete. Use `puppetSound` instead.

## soundDevice

### Syntax

the soundDevice

### Description

System property; allows the sound mixing device to be set while the movie plays. The possible settings are the devices listed in `soundDeviceList`.

Several sound devices can be referenced. The various sound devices for Windows have different advantages.

- **MacroMix (Windows)**—The lowest common denominator for Windows playback. This device functions on any Windows computer, but its latency is not as good as that of other devices.
- **QT3Mix (Windows)**—Mixes sound with QuickTime audio and possibly with other applications if they use DirectSound. This device requires that QuickTime be installed and has better latency than MacroMix.
- **MacSoundManager (Macintosh)**—The only sound device available on the Macintosh.

### Example

The following statement sets the sound device to the MacroMix for a Windows computer. If the newly assigned device fails, the `soundDevice` property is not changed.

```
set the soundDevice = "MacroMix"
```

### See also

`soundDeviceList`

## soundDeviceList

### Syntax

the soundDeviceList

### Description

System property; creates a linear list of sound devices available on the current computer.

For the Macintosh, this property lists one device, `MacSoundManager`.

This property can be tested but not set.

### Example

This statement displays a typical sound device list on a Windows computer:

```
Put the soundDeviceList  
--["QT3Mix", "MacroMix", "DirectSound"]
```

### See also

`soundDevice`

## soundEnabled

### Syntax

the soundEnabled

**Description**

System property; determines whether the sound is on (TRUE, default) or off (FALSE).

When you set this property to FALSE, the sound is turned off, but the volume setting is not changed.

This property can be tested and set.

**Example**

This statement sets `soundEnabled` to the opposite of its current setting; it turns the sound on if it is off and turns it off if it is on:

```
the soundEnabled = not(the soundEnabled)
```

**See also**

`soundLevel`, `volume (sound channel)`, `volume (sprite property)`

## sound fadeIn

**Syntax**

```
sound(whichChannel).fadeIn()  
sound(whichChannel).fadeIn(milliseconds)
```

```
sound fadeIn whichChannel  
sound fadeIn whichChannel, ticks
```

**Description**

Command; fades in a sound in the specified sound channel over a specified period of time.

The `sound(whichChannel).fadeIn()` syntax performs the fade over a specified number of frames or milliseconds.

- When *milliseconds* is specified, the fade in occurs evenly over that period of time.
- When *milliseconds* is not specified, the default number of ticks is calculated as  $15 * (1000 / (\text{tempo setting}))$  based on the tempo setting for the first frame of the fade in.

The `sound fadeIn whichChannel` syntax performs the fade over a specified number of frames or ticks. A tick is a 60th of a second.

- When *ticks* is specified, the fade in occurs evenly over that period of time.
- When *ticks* is not specified, the default number of ticks is calculated as  $15 * (60 / (\text{tempo setting}))$  based on the tempo setting for the first frame of the fade in.

The fade in continues at a predetermined rate until the time has elapsed, or until the sound in the specified channel changes.

**Example**

This statement fades in the sound in channel 1 over 5 seconds:

```
sound(1).fadeIn(5000)
```

**See also**

`sound fadeOut`, `fadeTo()`



## sound fadeOut

### Syntax

```
sound(whichChannel).fadeOut()  
sound(whichChannel).fadeOut(milliseconds)
```

```
sound fadeOut whichChannel  
sound fadeOut whichChannel, ticks
```

### Description

Command; fades out a sound in the specified sound channel over a specified period of time.

The `sound(whichChannel).fadeOut()` syntax performs the fade over a specified number of frames or milliseconds.

- When *milliseconds* is specified, the fade in occurs evenly over that period of time.
- When *milliseconds* is not specified, the default number of ticks is calculated as  $15 * (1000 / (\text{tempo setting}))$  based on the tempo setting for the first frame of the fade out.

The `sound fadeOut whichChannel` syntax performs the fade over a specified number of frames or ticks. A tick is a 60th of a second.

- When *ticks* is specified, the fade in occurs evenly over that period of time.
- When *ticks* is not specified, the default number of ticks is calculated as  $15 * (60 / (\text{tempo setting}))$  based on the tempo setting for the first frame of the fade out.

The fade out continues at a predetermined rate until the time has elapsed, or until the sound in the specified channel changes.

If the sound is stopped before it reaches the minimum volume, it remains at the level it was stopped at, causing subsequent playback to be at this volume. Be sure to allow the sound to finish fading completely.

**Note:** You may want to use the `volume` sound property to create a custom sound fade to allow more control over the actual volume of the channel.

### Example

This statement fades out the sound in channel 1 over 5 seconds:

```
sound(1).fadeOut(5000)
```

### See also

`puppetSound`, `sound fadeIn`, `volume` (sprite property), `fadeTo()`

## soundKeepDevice

### Syntax

```
the soundKeepDevice
```

### Description

System property; for Windows only, prevents the sound driver from unloading and reloading each time a sound needs to play. The default value is `TRUE`.

You may need to set this property to `FALSE` before playing a sound to ensure that the sound device is unloaded and made available to other applications or processes on the computer after the sound has finished.

Setting this property to `FALSE` may adversely affect performance if sound playback is used frequently throughout the Director application.

This property can be tested and set.

#### **Example**

This statement sets the `soundKeepDevice` property to `FALSE`:

```
set the soundKeepDevice = FALSE
```

## **soundLevel**

#### **Syntax**

```
the soundLevel
```

#### **Description**

System property; sets the volume level of the sound played through the computer's speaker. Possible values range from 0 (no sound) to 7 (the maximum, default).

In Windows, the system sound setting combines with the volume control of the external speakers. Thus, the actual volume that results from setting the sound level can vary. Avoid setting the `soundLevel` property unless you are sure that the result is acceptable to the user. It is better to set the individual volumes of the channels and sprites with `volume of sound`, `volume of member`, and `volume of sprite`.

These values correspond to the settings in the Macintosh Sound control panel. Using this property, Lingo can change the sound volume directly or perform some other action when the sound is at a specified level.

This property can be tested and set.

To see an example of `soundLevel` used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

#### **Examples**

This statement sets the variable `oldSound` equal to the current sound level:

```
oldSound = the soundLevel
```

This statement sets the sound level to 5:

```
the soundLevel = 5
```

#### **See also**

`soundEnabled`, `volume (sound channel)`

## **the soundMixMedia**

#### **Syntax**

```
the soundMixMedia
```

#### **Description**

This global property enables Flash cast members to mix their sound with sounds in the score sound channels when it is set to `TRUE`. When it is set to `FALSE`, these sounds will not be mixed and must be played at separate times. This property defaults to `TRUE` for movies made with Director 7 and later and `false` for earlier ones.

This property is valid only on Windows. When the `soundMixMedia` is `TRUE`, Director takes over the mixing and playback of sounds from Flash cast members. It is possible that slight differences may occur in the way Flash sounds play back. To hear the Flash sounds exactly they would be rendered in Flash, set this property to `FALSE`.

## sound playFile

### Syntax

```
sound playFile whichChannel, whichFile
```

### Description

Command; plays the AIFF, SWA, AU, or WAV sound located at *whichFile* in the sound channel specified by *whichChannel*. For the sound to be played properly, the correct MIX Xtra must be available to the movie, usually in the Xtras folder of the application.

When the sound file is in a different folder than the movie, *whichFile* must specify the full path to the file.

To play sounds obtained from a URL, it's usually a good idea to use `downloadNetThing` or `preloadNetThing()` to download the file to a local disk first. This approach can minimize problems that may occur while the file is downloading.

The `sound playFile` command streams files from disk rather than playing them from RAM. As a result, using the `sound playFile` command when playing digital video or when loading cast members into memory can cause conflicts when the computer tries to read the disk in two places at once.

### Examples

This statement plays the file named Thunder in channel 1:

```
sound playFile 1, "Thunder.wav"
```

This statement plays the file named Thunder in channel 3:

```
sound playFile 3, the moviePath & "Thunder.wav"
```

### See also

`sound stop`

## sound stop

### Syntax

```
sound(whichChannel).stop()  
sound stop whichChannel
```

### Description

Command; stops the sound playing in the specified channel.

The `sound stop` command was used in earlier versions of Director. For best results, use the `puppetSound` command instead.

### Examples

These statements stop any sound playing in sound channel 1:

```
sound(1).stop()
```

This statement checks whether a sound is playing in sound channel 1 and, if it is, stops the sound:

```
if soundBusy(1) then sound(1).stop()
```

### See also

`puppetSound`, `soundBusy()`

## source

### Syntax

```
sprite(whichSprite).camera.backdrop[backdropIndex].source  
member(whichCastmember).camera(whichCamera).backdrop  
[backdropIndex].source  
sprite(whichSprite).camera.overlay[overlayIndex].source  
member(whichCastmember).camera(whichCamera).overlay  
[overlayIndex].source
```

### Description

3D backdrop and overlay property; allows you to get or set the texture to use as the source image for the overlay or backdrop.

### Example

This statement sets the source of backdrop 1 to the texture Cedar:

```
sprite(3).camera.backdrop[1].source =  
    sprite(3).member.texture("Cedar")
```

### See also

bevelDepth, overlay

## sourceFileName

### Syntax

```
flashCastMember.sourceFileName
```

### Description

Flash cast member property; specifies the pathname of the FLA source file to be used during launch-and-edit operations. This property can be tested and set. The default is an empty string.

### Example

This Lingo sets the sourceFileName of the Flashcast member “SWF” to C:\FlashFiles\myFile fla:

```
member("SWF").sourceFileName = "C:\FlashFiles\myFile.fla"
```

## sourceRect

### Syntax

```
window whichWindow.sourceRect  
the sourceRect of window whichWindow
```

### Description

Window property; specifies the original Stage coordinates of the movie playing in the window specified by *whichWindow*.

This property is useful for returning a window to its original size and position after it has been dragged or its rectangle has been set.

**Example**

This statement displays the original coordinates of the Stage named `Control_panel` in the Message window:

```
put window("Control_panel").sourceRect
```

**See also**

`drawRect`, `rect (camera)`

## SPACE

**Syntax**

SPACE

**Description**

Constant; read-only, value that represents the space character.

**Example**

This statement displays “Age Of Aquarius” in the Message window:

```
put "Age"&SPACE&"Of"&SPACE&"Aquarius"
```

## specular (light)

**Syntax**

```
member(whichCastmember).light(whichLight).specular
```

**Description**

3D light property; allows you to get or set whether specularity is on (`TRUE`) or off (`FALSE`). Specularity refers to the ability to have a highlight appear on a model where the light hitting the model is reflected toward the camera. The shininess of the model's shader determines how large the specular portion of the highlight is. The value for this property is ignored for ambient lights. The default value for this property is `TRUE`.

**Note:** Turning off this property may increase performance.

**Example**

The following statement sets the specular property of the light `omni2` to `FALSE`. This light does not cause highlights. If this is the only light currently shining in the scene, there will be no specular highlights on any of the shaders in the scene.

```
member("3d world").light("omni2").specular = FALSE
```

**See also**

`silhouettes`, `specularLightMap`

## specular (shader)

### Syntax

`member(whichCastmember).shader(whichShader).specular`

### Description

3D standard shader property; allows you get or set the specular color of a given shader. The specular color is the color of the highlight generated when specularity is turned on. There must be lights in the scene with the specular property set to `TRUE`, for this property to have a visible effect. The specular color is influenced by the color of the lights illuminating the object. If the specular color is white but the color of a light is red, there will be a red specular highlight appearing on the object. The default value for this property is `rgb(255, 255, 255)` which is white.

All shaders have access to the `#standard` shader properties; in addition to these standard shader properties shaders of the types `#engraver`, `#newsprint`, and `#painter` have properties unique to their type. For more information, see `newShader`.

### Example

```
put member("scene").shader("plutomat").specular
-- rgb(11, 11, 11)
```

### See also

`silhouettes`, `specular (light)`, `specularColor`, `emissive`

## specularColor

### Syntax

`member(whichCastmember).specularColor`

### Description

3D cast member property; allows you to get or set the RGB value of the specular color of the first shader in the cast member. The first shader in the cast member's shader palette is always the default shader. This and all other 3D cast member properties are saved with the cast member and are restored the next time you open the movie. The default value for this property is `rgb(255, 255, 255)` which is white.

### Example

The following statement sets the specular color of the first shader in the cast member `Scene` to `rgb(255,0,0)`. It is equivalent to `member("Scene").shader[1].specular=rgb(255,0,0)`. However, that syntax won't save the new value with the cast member when the movie is saved. Only `member.specularColor` will save the new color value.

```
member("Scene").specularColor = rgb(255, 0, 0)
```

### See also

`silhouettes`, `specular (light)`, `specular (shader)`

# specularLightMap

## Syntax

```
member(whichCastmember).shader(whichShader).specularLightMap  
member(whichCastmember).model(whichModel).shader.specularLightMap  
member(whichCastmember).model(whichModel).shaderList\  
[shaderListIndex].specularLightMap
```

## Description

3D standard shader property; allows you to get or set the fifth texture layer of a given standard shader. This property is ignored if the `toon` modifier is applied to the model resource.

The values that can be set are as follows:

- `textureModeList[5] = #specular`
- `blendFunctionList[5] = #add`
- `blendFunctionList[1] = #replace`
- `default = void`

This helper property provides a simple interface for setting up a common use of specular light mapping.

All shaders have access to the `#standard` shader properties; in addition to these standard shader properties shaders of the types `#engraver`, `#newsprint`, and `#painter` have properties unique to their type. For more information, see the `newShader`.

## Example

This statement sets the texture `Oval` as the `specularLightMap` of the shader used by the model `GlassBox`:

```
member("3DPlanet").model("GlassBox").shader.specularLightMap = \  
member("3DPlanet").texture("Oval")
```

## See also

`diffuseLightMap`

# spotAngle

## Syntax

```
member(whichCastmember).light(whichLight).spotAngle
```

## Description

3D property; allows you to get or set the angle of the light projection cone. Light that is falls outside of the angle specified for this property, contributes no intensity. This property takes float value between 0.0 and 180.00, and has a default value of 90.0. The float value you specify corresponds to half the angle; for instance if you wish to specify a 90° angle you would pass a value of 45.0.

## Example

This statement sets the `spotAngle` property of the light `unidirectional` to 8. The angle of the light projection cone will be 16°:

```
member("3d world").light("unidirectional").spotAngle = 8
```



## spotDecay

### Syntax

```
member(whichCastmember).light(whichLight).spotDecay
```

### Description

3D light property; allows you get or set whether a spotlight's intensity falls off with the distance from the camera. The default value for this property is FALSE.

### Example

The following statement sets the `spotDecay` property of light 1 to TRUE. Models that are farther away from light 1 will be less brightly lit than models that are closer to it.

```
member("Scene").light[1].spotDecay = TRUE
```

## sprite

### Syntax

```
sprite(whichSprite).property  
the property of sprite whichSprite
```

### Description

Keyword; tells Lingo that the value specified by *whichSprite* is a sprite channel number. It is used with every sprite property.

A sprite is an occurrence of a cast member in a sprite channel of the Score.

This term has special meaning in the Director player for Java. Don't use the term `sprite` in Java code that you embed within a Lingo script.

### Examples

This statement sets the variable named `horizontal` to the `locH` of sprite 1:

```
horizontal = sprite(1).loc
```

This statement displays the current member in sprite channel 100 in the Message window:

```
put sprite (100).member
```

### See also

`puppetSprite`, `spriteNum`

## sprite...intersects

### Syntax

```
sprite(sprite1).intersects(sprite2)  
sprite sprite1 intersects sprite2
```

### Description

Keyword; operator that compares the position of two sprites to determine whether the quad of *sprite1* touches (TRUE) or does not touch (FALSE) the quad of *sprite2*.

If both sprites have matte ink, their actual outlines, not the quads, are used. A sprite's outline is defined by the nonwhite pixels that make up its border.

This is a comparison operator with a precedence level of 5.

**Note:** The dot operator is required whenever `sprite1` is not a simple expression—that is, one that contains a math operation.

### Example

This statement checks whether two sprites intersect and, if they do, changes the contents of the field cast member Notice to “You placed it correctly.”:

```
if sprite i intersects j then put "You placed it correctly." into member "Notice"
```

### See also

`sprite...within, quad`

## sprite...within

### Syntax

```
sprite(sprite1).within(sprite2)  
sprite sprite1 within sprite2
```

### Description

Keyword; operator that compares the position of two sprites and determines whether the quad of *sprite1* is entirely inside the quad of *sprite2* (TRUE) or not (FALSE).

If both sprites have matte ink, their actual outlines, not the quads, are used. A sprite’s outline is defined by the nonwhite pixels that make up its border.

This is a comparison operator with a precedence level of 5.

**Note:** The dot operator is required whenever `sprite1` is not a simple expression—that is, one that contains a math operation.

### Example

This statement checks whether two sprites intersect and calls the handler `doInside` if they do:

```
if sprite(3).within(2) then doInside
```

### See also

`sprite...intersects, quad`

## spriteNum

### Syntax

```
spriteNum  
the spriteNum of me
```

### Description

Sprite property; determines the channel number the behavior’s sprite is in and makes it available to any behaviors. Simply declare the property at the top of the behavior, along with any other properties the behavior may use.

If you use an `on new` handler to create an instance of the behavior, the script’s `on new` handler must explicitly set the `spriteNum` property to the sprite’s number. This provides a way to identify the sprite the script is attached to. The sprite’s number must be passed to the `on new` handler as an argument when the `on new` handler is called.

## Examples

In this handler, the `spriteNum` property is automatically set for script instances that are created by the system:

```
property spriteNum
```

```
on mouseDown me
    sprite(spriteNum).member = member("DownPict")
end
```

This handler uses the automatic value inserted into the `spriteNum` property to assign the sprite reference to a new property variable `pMySpriteRef`, as a convenience:

```
property spriteNum, pMySpriteRef
```

```
on beginSprite me
    pMySpriteRef = sprite(me.spriteNum)
end
```

This approach allows the use of the reference `pMySpriteRef` later in the script, with the handler using the syntax

```
currMember = pMySpriteRef.member
```

instead of the following syntax which is somewhat longer:

```
currMember = sprite(spriteNum).member
```

This alternative approach is merely for convenience, and provides no different functionality.

## See also

```
on beginSprite, on endSprite, currentSpriteNum
```

# spriteSpaceToWorldSpace

## Syntax

```
sprite(whichSprite).camera.spriteSpaceToWorldSpace(loc)
sprite(whichSprite).camera(index).spriteSpaceToWorldSpace(loc)
```

## Description

3D command; returns a world-space position that is found on the specified camera's projection plane that corresponds to a location within the referenced sprite. The location specified by *loc* should be a point relative to the sprite's upper-left corner.

The projection plane is defined by the camera's X and Y axes, and is at a distance in front of the camera such that one pixel represents one world unit of measurement. It is this projection plane that is used for the sprite display on stage.

The `camera.spriteSpaceToWorldSpace()` form of this command is a shortcut for using `camera(1).spriteSpaceToWorldSpace()`.

All cameras that are used by the referenced sprite will respond to the `spriteSpaceToWorldSpace` command as if their display rect is the same size as the sprite.

**Example**

This statement shows that the point (50, 50) within sprite 5 is equivalent to the vector (-1993.6699, 52.0773, 2263.7446) on the projection plane of the camera of sprite 5:

```
put sprite(5).camera.spriteSpaceToWorldSpace(point(50, 50))
-- vector(-1993.6699, 52.0773, 2263.7446)
```

**See also**

worldSpaceToSpriteSpace, *rect (camera)*, camera

## sqrt()

**Syntax**

```
sqrt(number)
the sqrt of number
```

**Description**

Math function; returns the square root of the number specified by *number*, which is either a floating-point number or an integer rounded to the nearest integer.

The value must be a decimal number greater than 0. Negative values return 0.

**Examples**

This statement displays the square root of 3.0 in the Message window:

```
put sqrt(3.0)
-- 1.7321
```

This statement displays the square root of 3 in the Message window:

```
put sqrt(3)
-- 2
```

**See also**

floatPrecision

## stage

**Syntax**

```
the stage
```

**Description**

System property; refers to the main movie.

This property is useful when using the `tell` command to send a message to the main movie from a child movie.

**Examples**

This statement causes the main Stage movie to go to the marker named Menu. This statement can be used in a movie in a window (MIAW):

```
tell the Stage to go to "Menu"
```

This statement displays the current setting for the Stage:

```
put the stage.rect
--rect (0, 0, 640, 480)
```

## stageBottom

### Syntax

the stageBottom

### Description

Function; along with stageLeft, stageRight, and stageTop, indicates where the Stage is positioned on the desktop. It returns the bottom vertical coordinate of the Stage relative to the upper left corner of the main screen. The height of the Stage in pixels is determined by the stageBottom - the stageTop.

When the movie plays back as an applet, the stageBottom property is the height of the applet in pixels.

This function can be tested but not set.

### Example

These statements position sprite 3 a distance of 50 pixels from the bottom edge of the Stage:

```
stageHeight = the stageBottom - the stageTop  
sprite(3).locV = stageHeight - 50
```

Sprite coordinates are expressed relative to the upper left corner of the Stage. For more information, see *Using Director*.

### See also

stageLeft, stageRight, stageTop, locH, locV

## stageColor

### Syntax

the stageColor

### Description

System property; determines the color of the movie background for index color only.

Use bgColor for more accurate, reliable, and flexible stage color specification with RGB values.

The value of the stageColor property ranges from 0 to 255 for 8-bit index color, and from 0 to 15 for 4-bit color. You can click a color in the color palette to see that color's index number in the lower left corner of the window. Setting the stageColor property in a Lingo script is equivalent to choosing the Stage color from the pop-up palette in the panel window.

**Note:** For compatibility when playing back as a Java applet, use the bgColor property to define the color as an RGB value.

### Examples

This statement sets the variable oldColor to the index number of the current Stage color:

Related Lingo:

```
oldColor = the stageColor
```

This statement sets the Stage color to the color assigned to chip 249 on the current palette:

Related function:

the stageColor = 249

**See also**

backColor, bgColor, foreColor, color()

## stageLeft

**Syntax**

the stageLeft

**Description**

Function; along with stageRight, stageTop, and stageBottom, indicates where the Stage is positioned on the desktop. It returns the left horizontal coordinate of the Stage relative to the upper left corner of the main screen. When the Stage is flush with the left side of the main screen, this coordinate is 0.

When the movie plays back as an applet, the stageLeft property is 0, which is the location of the left side of the applet.

This property can be tested but not set.

Sprite coordinates are expressed relative to the upper left corner of the Stage.

**Example**

This statement checks whether the left edge of the Stage is beyond the left edge of the screen and calls the handler leftMonitorProcedure if it is:

```
if the stageLeft < 0 then leftMonitorProcedure
```

**See also**

stageBottom, stageRight, stageTop, locH, locV

## stageRight

**Syntax**

the stageRight

**Description**

Function; along with stageLeft, stageTop, and stageBottom, indicates where the Stage is positioned on the desktop. It returns the right horizontal coordinate of the Stage relative to the upper left corner of the main screen's desktop. The width of the Stage in pixels is determined by the stageRight - the stageLeft.

When the movie plays back as an applet, the stageRight property is the width of the applet in pixels.

This function can be tested but not set.

Sprite coordinates are expressed relative to the upper left corner of the Stage.

### Example

These two statements position sprite 3 a distance of 50 pixels from the right edge of the Stage:

```
stageWidth = the stageRight - the stageLeft  
sprite(3).locH = stageWidth - 50
```

### See also

stageLeft, stageBottom, stageTop, locH, locV

## stageToFlash()

### Syntax

```
sprite(whichFlashSprite).stageToFlash(pointOnDirectorStage)  
stageToFlash (sprite whichFlashSprite, pointOnDirectorStage)
```

### Description

Function; returns the coordinate in a Flash movie sprite that corresponds to a specified coordinate on the Director Stage. The function both accepts the Director Stage coordinate and returns the Flash movie coordinate as Director point values: for example, point (300,300).

Flash movie coordinates are measured in Flash movie pixels, which are determined by the original size of the movie when it was created in Flash. Point (0,0) of a Flash movie is always at its upper left corner. (The cast member's `originPoint` property is not used to calculate movie coordinates; it is used only for rotation and scaling.)

The `stageToFlash()` function and the corresponding `flashToStage()` function are helpful for determining which Flash movie coordinate is directly over a Director Stage coordinate. For both Flash and Director, point (0,0) is the upper left corner of the Flash Stage or Director Stage. These coordinates may not match on the Director Stage if a Flash sprite is stretched, scaled, or rotated.

### Example

The following handler checks to see if the mouse pointer (whose location is tracked in Director Stage coordinates) is over a specific coordinate (130,10) in a Flash movie sprite in channel 5. If the pointer is over that Flash movie coordinate, the script stops the Flash movie.

```
on checkFlashRolllover  
  if sprite(5).stageToFlash(point(the mouseH,the mouseV)) = point(130,10) then  
    sprite(5).stop()  
  end if  
end
```

### See also

flashToStage()

## stageTop

### Syntax

```
the stageTop
```

### Description

Function; along with `stageBottom`, `stageLeft`, and `stageRight`, indicates where the Stage is positioned on the desktop. It returns the top vertical coordinate of the Stage relative to the upper left corner of the main screen's desktop. If the Stage is in the upper left corner of the main screen, this coordinate is 0.

When the movie plays back as an applet, the `stageTop` property is always 0, which is the location of the left side of the applet.

This function can be tested but not set.

Sprite coordinates are expressed relative to the upper left corner of the Stage.

#### Example

This statement checks whether the top of the Stage is beyond the top of the screen and calls the handler `upperMonitorProcedure` if it is:

```
if the stageTop < 0 then upperMonitorProcedure
```

#### See also

`stageLeft`, `stageRight`, `stageBottom`, `locH`, `locV`

## startAngle

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).  
startAngle  
modelResourceObjectReference.startAngle
```

### Description

3D property; when used with a model resource whose type is `#cylinder` or `#sphere`, this command allows you to both get and set the `startAngle` property of the referenced model resource, as a floating-point value from 0.0 to 360.0. The default value for this property is 0.0.

The `startAngle` property determines the starting sweep angle of the model resource, and works in conjunction with the `endAngle` property to draw spheres and cylinders. For example, if you want to make a half sphere, set `startAngle` to 0.0 and `endAngle` to 180.0.

### Example

The following statement sets the `startAngle` of the model resource `Sphere01` to 0.0. If its `endAngle` is set to 90, then only one quarter of any model that uses this model resource will appear.

```
put member("3D World").modelResource("Sphere01").startAngle  
-- 0.0
```

### See also

`endAngle`

## startFrame

### Syntax

```
sprite(whichSprite).startFrame
```

### Description

Function; returns the frame number of the starting frame of the sprite span.

This function is useful in determining the span in the Score that a particular sprite covers. It is available only in a frame that contains the sprite, and cannot be applied to sprites in different frames of the movie, nor is it possible to set this value.



**Example**

This statement displays the starting frame of the sprite in channel 5 in the Message window:

```
put sprite(5).startFrame
```

**See also**

`endFrame()`

## on startMovie

**Syntax**

```
on startMovie  
    statement(s)  
end
```

**Description**

System message and event handler; contains statements that run just before the playhead enters the first frame of the movie. The `startMovie` event occurs after the `prepareFrame` event and before the `enterFrame` event.

An `on startMovie` handler is a good place to put Lingo that initializes sprites in the first frame of the movie.

**Example**

This handler makes sprites invisible when the movie starts:

```
on startMovie  
    repeat with counter = 10 to 50  
        sprite(counter).visible = 0  
    end repeat  
end startMovie
```

**See also**

`on prepareMovie`

## starts

**Syntax**

```
string1 starts string2
```

**Description**

Operator; compares to determines whether *string1* starts with *string2* (TRUE or 1) or not (FALSE or 0).

The string comparison is not sensitive to case or diacritical marks; *a* and *À* are considered to be the same.

This is a comparison operator with a precedence level of 1.

**Example**

This statement reports in the Message window whether the word *Macromedia* starts with the string “Macro”:

```
put "Macromedia" starts "Macro"
```

The result is 1, which is the numerical equivalent of TRUE.

**See also**

`contains`

## startTime

**Syntax**

```
sprite(whichSprite).startTime  
the startTime of sprite whichSprite  
sound(channelNum).startTime
```

**Description**

Sprite and sound property; for digital video sprites, determines when the specified digital video sprite begins. The value of `startTime` is measured in ticks.

For digital video sprites, this property can be tested and set. Set the `startTime` before the digital video member begins playback.

For sound channels, this property indicates the start time of the currently playing or paused sound as set when the sound was queued. It cannot be set after the sound has been queued. If no value was supplied when the sound was queued, this property returns zero.

**Example**

This statement starts the digital video sprite in channel 5 at 100 ticks into the digital video:

```
sprite(5).startTime = 100
```

**See also**

`queue()`, `setPlaylist()`, `play()` (sound)

## startTimer

**Syntax**

```
startTimer
```

**Description**

Command; sets the `timer` property to 0 and resets all the accumulating timers for the `lastClick()`, `lastEvent()`, `lastKey`, and `lastRoll` functions to 0.

If multiple timers are required, you must create and manage your own. These can be properties in a behavior, a global list, or even a parent script. Typically, you use the `ticks` property to track time in this manner.

### Example

This handler sets the timer to 0 when a key is pressed:

```
on keyDown
    startTimer
end
```

### See also

`lastClick()`, `lastEvent()`, `lastKey`, `lastRoll`, `timeoutLength`, `timeoutMouse`, `timeoutPlay`, `timeoutScript`, `timer`

## state (3D)

### Syntax

`member(whichCastmember).state`

### Description

3D property; returns the current state of the referenced member in the streaming and loading process. This property refers to the initial file import or the last file load requested.

The `state` property of the member determines what, if any, 3D Lingo can be performed on the cast member.

This property can have any of the following values:

- 0—indicates that the member is currently not loaded and therefore no 3D media are available. No 3D Lingo should be performed on the member.
- 1—indicates that the media loading has begun.
- 2—indicates that the member's initial load segment is loaded. All objects with a stream priority of zero, as set upon creation of the model file, will be loaded at this time, because they are part of the initial load segment. You can perform most 3D Lingo associated with objects that have a load priority of zero. Do not use the `loadFile` and `resetWorld` commands during this state.
- 3—indicates that all the additional media of the member are being loaded and decompressed. Most 3D Lingo can be performed at this point. Do not use the `loadFile` and `resetWorld` commands during this state.
- 4—indicates that all of the member's media have been loaded and all decompression is complete. All 3D Lingo can now be performed on the cast member.
- -1—indicates that an undefined error occurred during the media streaming process. Because the error may have occurred at any point during the loading process, the state of the cast member is not reliable.

In general, avoid using Lingo on 3D cast members with a current state lower than 3.

### Example

This statement shows that the cast member named `PartyScene` has finished loading and preparing for playback, and no errors occurred during the load:

```
put member("PartyScene").state
-- 4
```

## state (Flash, SWA)

### Syntax

`member(whichCastMember).state`  
state of member *whichCastMember*

### Description

Cast member property; for Shockwave Audio (SWA) streaming cast members and Flash movie cast members, determines the current state of the streaming file. The properties `streamName`, `URL`, and `preLoadTime` can be changed only when the SWA sound is stopped.

The following properties for the SWA file return meaningful information only after the file begins streaming: `cuePointNames`, `cuePointTimes`, `currentTime`, `duration`, `percentPlayed`, `percentStreamed`, `bitRate`, `sampleRate`, and `numChannels`.

For SWA streaming cast members, the following values are possible:

- 0—Cast streaming has stopped.
- 1—The cast member is reloading.
- 2—Preloading ended successfully.
- 3—The cast member is playing.
- 4—The cast member is paused.
- 5—The cast member has finished streaming.
- 9—An error occurred.
- 10—There is insufficient CPU space.

For Flash movie cast members, this property returns a valid value only when the Director movie is running. The following values are possible:

- 0—The cast member is not in memory.
- 1—The header is currently loading.
- 2—The header has finished loading.
- 3—The cast member's media is currently loading.
- 4—The cast member's media has finished loading.
- -1—An error occurred.

This property can be tested but not set.

### Example

This statement issues an alert if an error is detected for the SWA streaming cast member:

```
on mouseDown
  if member("Ella Fitzgerald").state = 9 then
    alert "Sorry, can't find an audio file to stream."
  end if
end
```

### Example

The following frame script checks to see if a Flash movie cast member named Intro Movie has finished streaming into memory. If it hasn't, the script reports in the Message window the current state of the cast member and keeps the playhead looping in the current frame until the movie finishes loading into memory.

```
on exitFrame
  if member("Intro Movie").percentStreamed < 100 then
    put "Current download state:" && member("Intro Movie").state
    go the frame
  end if
end
```

### See also

`clearError`, `getError()`

## state (RealMedia)

### Syntax

```
sprite(whichSprite).state  
member(whichCastmember).state
```

### Description

RealMedia sprite or cast member property; returns the current state of the RealMedia stream, expressed as an integer in the range 1 to 4. Each state value corresponds to a specific point in the streaming process. This property is dynamic during playback and can be tested but not set.

The streaming process is initiated when the playhead enters the span of the RealMedia sprite in the Score, the `play` method is invoked on a RealMedia sprite or cast member, or a user clicks the Play button in the RealMedia viewer. Calling this property returns a numeric value indicating the state of the streaming process for the RealMedia cast member. For each state there is one or more corresponding `mediaStatus` property value; each `mediaStatus` value is observed only in one state. For example, the `mediaStatus` property values `#seeking` and `#buffering` are present only when the value of `state` is 3.

The value of the `state` property provides important information in terms of performing Lingo on a cast member. If `member.state` is less than 2, some of the Lingo properties may be incorrect, and as a result, any Lingo relying on property data would be incorrect. When `member.state` is greater than or equal to 2 and less than 4, the RealMedia cast member is not displayed, but all the Lingo properties and methods have well-defined values and can be used to perform Lingo operations on the cast member.

When the streaming process is initiated, the `state` property cycles through the following states, unless an error (-1) occurs, which prevents the streaming process from starting:

**-1 (error)** indicates that there is something wrong, possibly a leftover error from the previous RealMedia stream. You may get more information by checking the `lastError` property. This state is the equivalent of `#error` for the `mediaStatus` property.

**0 (closed)** indicates that streaming has not begun, or that cast member properties are in initial states or are copies from an earlier playing of the cast member. This state is the equivalent of `#closed` for the `mediaStatus` property.

**1 (connecting)** indicates that streaming has begun but is in the very early stages of connecting to the server, and there is not enough information available locally to do anything with the cast member. This state is the equivalent of `#connecting` for the `mediaStatus` property.

**2 (open)** indicates that the Lingo properties have been refreshed from the actual stream. When state is greater than or equal to 2, the `height`, `width`, and `duration` properties of the `RealMedia` stream are known. This state is transitory and quickly changes to state 3. This state is the equivalent of `#opened` for the `mediaStatus` property.

**3 (seeking or buffering)** indicates that all of the `RealMedia` cast member's Lingo properties are current, but the cast member is not quite ready to play. The Stage or `RealMedia` viewer displays a black rectangle or the `RealNetworks` logo. If this state is the result of rebuffering due to network congestion, the `state` value quickly changes back to 4 (playing). This state is the equivalent of `#buffering` or `#seeking` for the `mediaStatus` property.

**4 (playing)** indicates that the `RealMedia` stream is playing (or paused) without problems or errors. This is the state during normal playback. This state is the equivalent of `#playing` or `#paused` for the `mediaStatus` property.

### Examples

The following examples show that the state of streams in sprite 2 and the cast member `Real` is 0, which is closed:

```
put sprite(2).state
-- 0

put member("Real").state
-- 0
```

### See also

`mediaStatus`, `percentBuffered`, `lastError`

## static

### Syntax

```
sprite(whichFlashSprite).static
the static of sprite whichFlashSprite
member(whichFlashMember).static
the static of member whichFlashMember
```

### Description

Cast member property and sprite property; controls playback performance of a Flash movie sprite depending on whether the movie contains animation. If the movie contains animation (`FALSE`, default), the property redraws the sprite for each frame; if the movie doesn't contain animation (`TRUE`), the property redraws the sprite only when it moves or changes size.

This property can be tested and set.

**Note:** Set the `static` property to `TRUE` only when the Flash movie sprite does not intersect other moving Director sprites. If the Flash movie intersects moving Director sprites, it may not redraw correctly.

### Example

This sprite script displays in the Message window the channel number of a Flash movie sprite and indicates whether the Flash movie contains animation:

```
property spriteNum

on beginSprite me
  if sprite(spriteNum).static then
    animationType = "does not have animation."
  else
    animationType = "has animation."
  end if
  put "The Flash movie in channel" && spriteNum && animationType
end
```

## staticQuality

### Syntax

`staticQuality of sprite` *whichQTVRSprite*

### Description

QuickTime VR sprite property; specifies the codec quality used when the panorama image is static. Possible values are `#minQuality`, `#maxQuality`, and `#normalQuality`.

This property can be tested and set.

## status

### Syntax

`soundObject.status`  
the status of *soundObject*

### Description

Read-only property indicates the status of sound channel `channelNum`. Possible values include:

Status	Name	Meaning
0	Idle	No sounds are queued or playing.
1	Loading	A queued sound is being preloaded but is not yet playing.
2	Queued	The sound channel has finished preloading a queued sound but is not yet playing the sound.
3	Playing	A sound is playing.
4	Paused	A sound is paused.

### Example

This statement displays the current status of sound channel 2 in the Message window:

```
put sound(2).status
```

### See also

`isBusy()`, `pause()` (sound playback), `play()` (sound), `stop()` (sound)

## on stepFrame

### Syntax

```
on stepFrame
    statement(s)
end
```

### Description

System message and event handler; works in script instances in `actorList` because these are the only objects that receive `on stepFrame` messages. This event handler is executed when the playhead enters a frame or the Stage is updated.

An `on stepFrame` handler is a useful location for Lingo that you want to run frequently for a specific set of objects. Assign the objects to `actorList` when you want Lingo in the `on stepFrame` handler to run; remove the objects from `actorList` to prevent Lingo from running. While the objects are in `actorList`, the objects' `on stepFrame` handlers run each time the playhead enters a frame or the `updateStage` command is issued.

The `stepFrame` message is sent before the `prepareFrame` message.

Assign objects to `actorList` so they respond to `stepFrame` messages. Objects must have an `on stepFrame` handler to use this built-in functionality with `actorList`.

The `go`, `play`, and `updateStage` commands are disabled in an `on stepFrame` handler.

### Example

If the child object is assigned to `actorList`, the `on stepFrame` handler in this parent script updates the position of the sprite that is stored in the `mySprite` property each time the playhead enters a frame:

```
property mySprite

on new me, theSprite
    mySprite = theSprite
    return me
end

on stepFrame me
    sprite(mySprite).loc = point(random(640),random(480))
end
```

## stillDown

### Syntax

```
the stillDown
```

### Description

System property; indicates whether the user is pressing the mouse button (TRUE) or not (FALSE).

This function is useful within a `mouseDown` script to trigger certain events only after the `mouseUp` function.

Lingo cannot test `stillDown` when it is used inside a repeat loop. Use the `mouseDown` function inside repeat loops instead.



**Example**

This statement checks whether the mouse button is being pressed and calls the handler `dragProcedure` if it is:

```
if the stillDown then dragProcedure
```

**See also**

the `mouseDown` (system property)

## stop (Flash)

**Syntax**

```
sprite(whichFlashSprite).stop()  
stop sprite whichFlashSprite
```

**Description**

Flash command; stops a Flash movie sprite that is playing in the current frame.

**Example**

This frame script stops the Flash movie sprites playing in channels 5 through 10:

```
on enterFrame  
  repeat with i = 5 to 10  
    sprite(i).stop()  
  end repeat  
end
```

**See also**

`hold`

## stop (RealMedia)

**Syntax**

```
sprite(whichSprite).stop()  
member(whichCastmember).stop()
```

**Description**

RealMedia sprite or cast member method; stops playback of the media stream and resets `currentTime` to 0 and `percentBuffered` to 0. The `mediaStatus` value becomes `#stopped`.

Calling the `play` command after calling the `stop` command requires the stream to rebuffer, and playback begins at the beginning of the stream.

**Examples**

The following examples stop sprite 2 and the cast member Real from playing:

```
sprite(2).stop()  
member("Real").stop()
```

**See also**

`play (RealMedia)`, `pause (RealMedia)`, `stop (RealMedia)`, `mediaStatus`

## stop() (sound)

### Syntax

```
sound(channelNum).stop()  
stop(sound(channelNum))
```

### Description

Command; stops the currently playing sound in sound channel *channelNum*. Issuing a `play()` command begins playing the first sound of those that remain in the queue of the given sound channel.

To see an example of `stop() (sound)` used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This statement stops playback of the sound cast member currently playing in sound channel 1:

```
sound(1).stop()
```

### See also

`getPlaylist()`, `pause() (sound playback)`, `play() (sound)`, `playNext()`, `rewind()`

## stopEvent

### Syntax

```
stopEvent
```

### Description

Command; prevents Lingo from passing an event message to subsequent locations in the message hierarchy. Equivalent to the `dontPassEvent` command used in earlier versions of Director, this command also applies to sprite scripts.

Use the `stopEvent` command to stop the message in a primary event handler or a sprite script, thus making the message unavailable for subsequent sprite scripts.

By default, messages are available first to a primary event handler (if one exists) and then to any scripts attached to a sprite involved in the event. If more than one script is attached to the sprite, the message is available to each of the sprite's scripts. If no sprite script responds to the message, the message passes to a cast member script, frame script, and movie script, in that order.

The `stopEvent` command applies only to the current event being handled. It does not affect future events. The `stopEvent` command applies only within primary event handlers, handlers that primary event handlers call, or multiple sprite scripts. It has no effect elsewhere.

### Example

This statement shows the `mouseUp` event being stopped in a behavior if the global variable `grandTotal` is equal to 500:

```
global grandTotal  
  
on mouseUp me  
    if grandTotal = 500 then  
        stopEvent  
    end if  
end
```

Neither subsequent scripts nor other behaviors on the sprite receive the event if it is stopped in this manner.

**See also**

`pass`

## stop member

**Syntax**

```
member (whichCastMember ).stop()  
stop member (whichCastMember)
```

**Description**

Command; stops the playback of a Shockwave Audio (SWA) streaming cast member. When the cast member is stopped, the `state` member property equals 0.

For you to change properties such as `streamName`, `preLoadTime`, and `URL`, the SWA streaming cast member must be stopped.

**Example**

This statement stops the SWA cast member Big Band from playing:

```
member("Big Band").stop()
```

**See also**

`play member`, `pause member`

## on stopMovie

**Syntax**

```
on stopMovie  
    statement(s)  
end
```

**Description**

System message and event handler; contains statements that run when the movie stops playing.

An `on stopMovie` handler is a good place to put Lingo that performs cleanup tasks—such as closing resource files, clearing global variables, erasing fields, and disposing of objects—when the movie is finished.

An `on stopMovie` handler in a MIAW is called only when the movie plays through to the end or branches to another movie. It isn't called when the window is closed or when the window is deleted by the `forget window` command.

**Example**

This handler clears global variables and closes two resource files when the movie stops:

```
global gCurrentScore  
on stopMovie  
    set gCurrentScore = 0  
    closeResFile "Special Fonts"  
    closeResFile "Special Cursors"  
end
```

**See also**

`on prepareMovie`

## stopTime

### Syntax

`sprite(whichSprite).stopTime`  
the stopTime of sprite *whichSprite*

### Description

Sprite property; determines when the specified digital video sprite stops. The value of `stopTime` is measured in ticks.

This property can be tested and set.

### Example

This statement stops the digital video sprite in channel 5 at 100 ticks into the digital video:

```
sprite(5).stopTime = 100
```

## stream

### Syntax

`member(whichFlashSprite).stream(numberOfBytes )`  
`stream(member whichFlashSprite, numberOfBytes)`

### Description

Command; manually streams a portion of a specified Flash movie cast member into memory. You can optionally specify the number of bytes to stream as an integer value. If you omit the *numberOfBytes* parameter, Director tries to stream the number of bytes set by the cast member's `bufferSize` property.

The `stream` command returns the number of bytes actually streamed. Depending on a variety of conditions (such as network speed or the availability of the requested data), the number of bytes actually streamed may be less than the number of bytes requested.

You can always use the `stream` command for a cast member regardless of the cast member's `streamMode` property.

### Example

The following frame script checks to see if a linked Flash movie cast member has streamed into memory by checking its `percentStreamed` property. If the cast member is not completely loaded into memory, the script tries to stream 32,000 bytes of the movie into memory.

The script also saves the actual number of bytes streamed in a variable called `bytesReceived`. If the number of bytes actually streamed does not match the number of bytes requested, the script updates a text cast member to report the number of bytes actually received. The script keeps the playhead looping in the current frame until the cast member has finished loading into memory.

```
on exitFrame
  if member(10).percentStreamed < 100 then
    bytesReceived = member(10).stream(32000)
    if bytesReceived < 32000 then
      member("Message Line").text = "Received only" && bytesReceived \
        && "of 32,000 bytes requested."
      updateStage
    else
      member("Message Line").text = "Received all 32,000 bytes."
    end if
    go the frame
  end if
end
```

## streaming

### Syntax

```
member(whichMember).streaming
the streaming of member whichMember
```

### Description

QuickTime cast member property. When `TRUE`, allows QuickTime playing over the Internet to begin playing immediately while the member downloads to the user's computer. When `FALSE`, the member must download completely before playback will begin.

Defaults to `TRUE` for Director movies made with Director 7.02 and later. Defaults to `FALSE` for movies made with earlier versions of Director.

If the QuickTime member contains a text track with cue points, the text track must be set to preload in order for Director to make use of the cue points. You set the text track to preload using a video editor.

### Example

This statement sets the streaming of member `SunriseVideo` to `FALSE`, causing it to download completely before playing back in Shockwave or ShockMachine:

```
member("SunriseVideo").streaming = 0
```

## streamMode

### Syntax

`member(whichFlashMember).streamMode`  
the `streamMode` of member *whichFlashMember*

### Description

Flash cast member property; controls the way a linked Flash movie cast member is streamed into memory, as follows:

- `#frame` (default)—Streams part of the cast member each time the Director frame advances while the sprite is on the Stage.
- `#idle`—Streams part of the cast member each time an idle event is generated or at least once per Director frame while the sprite is on the Stage.
- `#manual`—Streams part of the cast member into memory only when the `stream` command is issued for that cast member.

This property can be tested and set.

### Example

This `startMovie` script searches the internal cast for Flash movie cast members and sets their `streamMode` properties to `#manual`:

```
on startMovie
  repeat with i = 1 to the number of members of castLib 1
    if member(i, 1).type = #flash then
      member(i, 1).streamMode = #manual
    end if
  end repeat
end
```

## streamName

### Syntax

`member(whichCastMember).streamName`  
the `streamName` of member *whichCastMember*

### Description

Shockwave Audio (SWA) cast member property; specifies a URL or filename for a streaming cast member. This property functions the same as the `URL` member property.

This property can be tested and set.

### Example

The following statement links the file `BigBand.swa` to an SWA streaming cast member. The linked file is on the disk `MyDisk` in the folder named `Sounds`.

```
member("SWAstream").streamName = "MyDisk/sounds/BigBand.swa"
```

## streamSize

### Syntax

`member(whichFlashMember).streamSize`  
the `streamSize` of member *whichFlashMember*

### Description

Cast member property; reports an integer value indicating the total number of bytes in the stream for the specified cast member. The `streamSize` property returns a value only when the Director movie is playing.

This property can be tested but not set.

### Example

The following frame script checks to see if a Flash movie cast member named Intro Movie has finished streaming into memory. If it hasn't, the script updates a field cast member to indicate the number of bytes that have been streamed (using the `bytesStreamed` member property) and the total number of bytes for the cast member (using the `streamSize` member property). The script keeps the playhead looping in the current frame until the movie finishes loading into memory.

```
on exitFrame
  if member("Intro Movie").percentStreamed < 100 then
    member("Message Line").text = string(member("Intro Movie").bytesStreamed) \
      && "of" && string(member("Intro Movie").streamSize) \
      && "bytes have downloaded so far."
    go to the frame
  end if
end
```

## streamSize (3D)

### Syntax

`member(whichCastmember).streamSize`

### Description

3D property; allows you to get the size of the data stream to be downloaded, from 0 to `maxInteger`. This command refers to the initial file import or the last `loadFile()` requested.

### Example

This statement shows that the last file load associated with the cast member Scene has a total size of 325300 bytes:

```
put member("Scene").streamSize
-- 325300
```

### See also

`bytesStreamed (3D)`, `percentStreamed (3D)`, `state (3D)`, `preLoad (3D)`

## on streamStatus

### Syntax

```
on streamStatus URL, state, bytesSoFar, bytesTotal, error
    statement(s)
end
```

### Description

System message and event handler; called periodically to determine how much of an object has been downloaded from the Internet. The handler is called only if `tellStreamStatus (TRUE)` has been called, and the handler has been added to a movie script.

The `on streamStatus` event handler has the following parameters:

---

<i>URL</i>	Displays the Internet address of the data being retrieved.
<i>state</i>	Displays the state of the stream being downloaded. Possible values are <code>Connecting</code> , <code>Started</code> , <code>InProgress</code> , <code>Complete</code> , and <code>Error</code> .
<i>bytesSoFar</i>	Displays the number of bytes retrieved from the network so far.
<i>bytesTotal</i>	Displays the total number of bytes in the stream, if known. The value may be 0 if the HTTP server does not include the content length in the MIME header.
<i>error</i>	Displays an empty string ("" ) if the download has not finished; OK ( <i>OK</i> ) if the download completed successfully; displays an error code if the download was unsuccessful.

---

These parameters are automatically filled in by Director with information regarding the progress of the download. The handler is called by Director automatically, and there is no way to control when the next call will be. If information regarding a particular operation is needed, call `getStreamStatus()`.

You can initiate network streams using Lingo commands, by linking media from a URL, or by using an external cast member from a URL. A `streamStatus` handler will be called with information about all network streams.

Place the `streamStatus` handler in a movie script.

### Example

This handler determines the state of a streamed object and displays the URL of the object:

```
on streamStatus URL, state, bytesSoFar, bytesTotal
    if state = "Complete" then
        put URL && "download finished"
    end if
end streamStatus
```

### See also

`getStreamStatus()`, `tellStreamStatus()`



## string()

### Syntax

`string(expression)`

### Description

Function; converts an integer, floating-point number, object reference, list, symbol, or other nonstring expression to a string.

### Examples

This statement adds 2.0 + 2.5 and inserts the results in the field cast member Total:

```
member("total").text = string(2.0 + 2.5)
```

This statement converts the symbol #red to a string and inserts it in the field cast member Color:

```
member("Color").text = string(#red)
```

### See also

`value()`, `stringP()`, `float()`, `integer()`, `symbol()`

## stringP()

### Syntax

`stringP(expression)`

### Description

Function; determines whether an expression is a string (TRUE) or not (FALSE).

The *P* in `stringP` stands for *predicate*.

### Examples

This statement checks whether 3 is a string:

```
put stringP("3")
```

The result is 1, which is the numeric equivalent of TRUE.

This statement checks whether the floating-point number 3.0 is a string:

```
put stringP(3.0)
```

Because 3.0 is a floating-point number and not a string, the result is 0, which is the numeric equivalent of FALSE.

### See also

`floatP()`, `ilk()`, `integerP()`, `objectP()`, `symbolP()`

## strokeColor

### Syntax

`member(whichCastMember).strokeColor`

### Description

Vector shape cast member property; indicates the color in RGB of the shape's framing stroke.

To see an example of `strokeColor` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

**Example**

This sets the `strokeColor` of cast member "line" to red:

```
member("line").strokeColor = rgb(255, 0, 0)
```

**See also**

`color()`, `fillColor`, `endColor`, `backgroundColor`

## strokeWidth

**Syntax**

```
member(whichCastMember).strokeWidth
```

**Description**

Vector shape cast member property; indicates the width, in pixels, of the shape's framing stroke.

The value is a floating-point number between 0 and 100 and can be tested and set.

To see an example of `strokeWidth` used in a completed movie, see the Vector Shapes movie in the Learning/Lingo Examples folder inside the Director application folder.

**Example**

The following code sets the `strokeWidth` of cast member "line" to 10 pixels:

```
member("line").strokeWidth = 10
```

## style

**Syntax**

```
member(whichCastmember).model(whichModel).toon.style  
member(whichCastmember).model(whichModel).shader.style  
member(whichCastmember).shader(whichShader).style
```

**Description**

3D toon modifier and painter shader property; indicates the way the toon modifier and painter shader apply color to a model. Possible values are as follows:

- `#toon` uses sharp transitions between colors.
- `#gradient` uses smooth transitions between colors. This is the default value.
- `#blackAndWhite` uses two-color black and white.

The number of colors used by the toon modifier and painter shader is set with the `colorSteps` property of the modifier or shader.

**Example**

The following statement sets the `style` property of the toon modifier for the model named Teapot to `#blackAndWhite`. The model will be rendered in two-color black and white.

```
member("Shapes").model("Teapot").toon.style = #blackAndWhite
```

## subdivision

### Syntax

`member(whichCastmember).model(whichModel).sds.subdivision`

### Description

3D `sds` modifier property; allows you to get or set the subdivision surfaces mode of operation. Possible values are as follows:

- `#uniform` specifies that the mesh is uniformly scaled up in detail, with each face subdivided the same number of times.
- `#adaptive` specifies that additional detail is added only when there are large surface orientation changes and only to those areas of the mesh that are currently visible.

The `sds` modifier cannot be used with the `inker` or `toon` modifiers, and caution should be used when using the `sds` modifier with the `lod` modifier. See the `sds` modifier entry for more information.

### Example

The following statement sets the `subdivision` property of the `sds` modifier of the model named `Baby` to `#adaptive`. `Baby`'s geometry will not be modified uniformly.

```
member("Scene").model("Baby").sds.subdivision = #adaptive
```

### See also

`sds` (modifier), `error`, `enabled (sds)`, `depth (3D)`, `tension`

## substituteFont

### Syntax

```
TextMemberRef.substituteFont(originalFont, newFont)  
substituteFont(textMemberRef, originalFont, newFont)
```

### Description

Textcastmember command; replaces all instances of *originalFont* with *newFont* in *textMemberRef*.

### Example

This script checks to see if the font `Bonneville` is available in a text cast member, and replaces it with `Arial` if it is not:

```
property spriteNum  
  
on beginSprite me  
  currMember = sprite(spriteNum).member  
  if currMember.missingFonts contains "Bonneville" then  
    currMember.substituteFont("Bonneville", "Arial")  
  end if  
end
```

### See also

`missingFonts`

## suspendUpdates

### Syntax

```
sprite(which3dSprite).suspendUpdates
```

### Description

3D sprite property; when set to `TRUE`, causes the sprite not to be updated as part of normal screen redraw operations. This can improve movie playback performance. Certain kinds of screen updates will still affect the sprite, such as those due to dragging another window over the sprite. When the `suspendUpdates` property is set to `FALSE`, the sprite is redrawn normally.

It is important to keep the `suspendUpdates` property set to `FALSE` while any element within the 3D sprite is being animated.

## swing()

### Syntax

```
WhichQTVRSprite.swing(pan, tilt, fieldOfView, speedToSwing)  
swing(whichQTVRSprite, pan, tilt, fieldOfView, speedToSwing)
```

### Description

QuickTime VR sprite function; swings a QuickTime 3 sprite containing a VR Pano around to the new view settings. The swing is a smooth “camera dolly” effect.

- *whichQTVRSprite*—Sprite number of the sprite with the QuickTime VR member.
- *pan*—New pan position, in degrees.
- *tilt*—New tilt, in degrees.
- *fieldOfView*—New field of view, in degrees.
- *speedToSwing*—Rate at which the swing should take place; specify an integer from 1 to 10 (slow to fast).

The function returns immediately, but the sprite continues to change view until it reaches the final view. The duration required to change to the final settings varies depending on machine type, size of the sprite rectangle, color depth of the screen, and other typical performance constraints.

To check if the swing has finished, check if the `pan` property of the sprite has arrived at the final value.

### Example

This very gradually adjusts the view of QTVR sprite 1 to a pan position of 300°, a tilt of -15°, and a field of view of 40°:

```
sprite(1).swing(300, -15, 40, 1)
```

### See also

`pan` (QTVR property)

## switchColorDepth

### Syntax

the switchColorDepth

### Description

System property; determines whether Director switches the monitor that the Stage occupies to the color depth of the movie being loaded (TRUE) or leaves the color depth of the monitor unchanged when a movie is loaded (FALSE). False is the default value.

When switchColorDepth is TRUE, nothing happens until a new movie is loaded.

Setting the monitor's color depth to that of the movie is good practice.

- When the monitor's color depth is set below that of the movie, resetting it to the color depth of the movie (assuming that the monitor can provide that color depth) helps maintain the movie's original appearance.
- When the monitor's color depth is higher than that of the movie, reducing the monitor's color depth plays the movie using the minimum amount of memory, loads cast members more efficiently, and causes animation to occur more quickly.

This property can be tested and set. The default value comes from the Reset Monitor to Movie's Color Depth option in the General Preferences dialog box.

### Examples

This statement sets the variable named `switcher` to the current setting of `switchColorDepth`:

```
switcher = the switchColorDepth
```

This statement checks whether the current color depth is 8-bit and turns the `switchColorDepth` property on if it is:

```
if the colorDepth = 8 then the switchColorDepth = TRUE
```

### See also

colorDepth

## symbol()

### Syntax

```
stringValue.symbol  
symbol(stringValue)
```

### Description

Function; takes a string, *stringValue*, and returns a symbol.

### Examples

This statement displays the symbol `#hello`:

```
put ("hello").symbol  
-- #hello
```

This statement displays the symbol `#goodbye`:

```
x = "goodbye"
put x.symbol
-- #goodbye
```

**See also**

`value()`, `string()`

## symbolP()

**Syntax**

```
Expression.symbolP
symbolP(expression)
```

**Description**

Function; determines whether the expression specified by *expression* is a symbol (TRUE) or not (FALSE).

The *P* in `symbolP` stands for *predicate*.

**Example**

This statement checks whether the variable `myVariable` is a symbol:

```
put myVariable.symbolP
```

**See also**

`ilk()`

## systemDate

**Syntax**

```
the systemDate
```

**Description**

System property; returns the current date in a standard date format and can be used in conjunction with other date operations for international and cross-platform date manipulation.

Math operations on the date are performed in days.

This property can be tested but not set.

**Example**

This script displays the current date being retrieved and then determines the date 90 days from the current date:

```
on ShowEndOfTrialPeriodDate
  set today = the systemDate
  set trialEndDate = today + 90
  put "The trial period for this software is over on"&&trialEndDate
end
```

**See also**

`date()` (system clock)

# TAB

## Syntax

TAB

## Description

Constant; represents the Tab key.

## Examples

This statement checks whether the character typed is the tab character and calls the handler `doNextField` if it is:

```
if the key = TAB then doNextField
```

These statements move the playhead forward or backward, depending on whether the user presses Tab or Shift-Tab:

```
if the key = TAB then
  if the shiftDown then
    go the frame -1
  else
    go the frame +1
  end if
end if
```

## See also

BACKSPACE, EMPTY, RETURN (constant)

# tabCount

## Syntax

*chunkExpression*.tabCount

## Description

Text cast member property; indicates how many unique tab stops are in the specified chunk expression of the text cast member.

The value is an integer equal to or greater than 0, and may be tested but not set.

# tabs

## Syntax

*member(whichTextMember)*.tabs

## Description

Text cast member property; this property contains a list of all the tab stops set in the text cast member.

Each element of the list contains information regarding that tab for the text cast member. The possible properties in the list are as follows:

---

#type	Can be #left, #center, #right, or #decimal.
#position	Integer value indicating the position of the tab in points.

---

You can get and set this property. When `tabs` is set, the `type` property is optional. If `type` is not specified, the tab type defaults to `#left`.

### Example

This statement retrieves and displays in the Message window all the tabs for the text cast member Intro credits:

```
put member("Intro credits").tabs
-- [[#type: #left, #position: 36], [#type: #Decimal, #position: 141], \
   [#type: #right, #position: 216]]
```

## tan()

### Syntax

*tan(angle)*

### Description

Math function; yields the tangent of the specified angle expressed in radians as a floating-point number.

### Example

The following function yields the tangent of  $\pi/4$ :

```
tan (PI/4.0) = 1
```

The  $\pi$  symbol cannot be used in a Lingo expression.

### See also

PI

## target

### Syntax

*timeoutObject.target*

### Description

Timeout object property; indicates the child object that the given *timeoutObject* will send its timeout events to. Timeout objects whose target property is `VOID` will send their events to a handler in a movie script.

This property is useful for debugging behaviors and parent scripts that use timeout objects.

### Example

This statement displays the name of the child object that will receive timeout events from the timeout object `timerOne` in the Message window:

```
put timeout("timerOne").target
```

### See also

`name (timeout property)`, `timeout()`, `timeoutHandler`, `timeoutList`



## targetFrameRate

### Syntax

```
sprite(which3dSprite).targetFrameRate
```

### Description

3D sprite property; determines the preferred number of frames per second to use when rendering a 3D sprite. The default value is 30 frames per second. The `targetFrameRate` property is only used if the `useTargetFrameRate` property is set to `TRUE`. If the `useTargetFrameRate` property is set to `TRUE`, Director will reduce the polygon count of the models in the sprite if necessary to maintain the specified frame rate.

### Example

These statements set the `targetFrameRate` property of sprite 3 to 45 and enforce the frame rate by setting the `useTargetFrameRate` property of the sprite to `TRUE`:

```
sprite(3).targetFrameRate = 45  
sprite(3).useTargetFrameRate = TRUE
```

### See also

`useTargetFrameRate`

## tell

### Syntax

```
tell whichWindow to statement(s)  
tell whichWindow  
    statement(s)  
end
```

### Description

Command; communicates statements to the window specified by *whichWindow*.

The `tell` command is useful for allowing movies to interact. It can be used within a main movie to send a message to a movie playing in a window, or to send a message from a movie playing in a window to the main movie. For example, the `tell` command can let a button in a control panel call a handler in a movie playing in a window. The movie playing in a window may react to the first movie handler by executing the handler. The movie playing in the window may interact with the main movie by sending a value back to the movie.

When you use the `tell` command to send a message to a movie playing in a window, identify the window object by using the full pathname or its number in `windowList`. If you use `windowList`, use the expression `getAt(the windowList, windowNum)`, where *windowNum* is a variable that contains the number of the window's position in the list. Because the opening and closing of windows may change the order of `windowList`, it is a good idea to store the full pathname as a global variable when referencing windows with `getAt` in `windowList`.

## Examples

A multiple-line `tell` command resembles a handler and requires an `end tell` statement:

```
global childMovie

tell window childMovie
  go to frame "Intro"
  the stageColor = 100
  sprite(4).member = member "Diana Ross"
  updateStage
end tell
```

When a message calls a handler, a value returned from the handler can be found in the global `result` property after the called handler is done. These statements send the `childMovie` window the message `calcBalance` and then return the result:

```
global childMovie

tell window childMovie to calcBalance
-- a handler name
myBalance = result()
-- return value from calcBalance handler
```

When you use the `tell` command to send a message from a movie playing in a window to the main movie, use the `stage` system property as the object name:

```
tell the stage to go frame "Main Menu"
```

When you use the `tell` command to call a handler in another movie, make sure that you do not have a handler by the same name in the same script in the local movie. If you do, the local script will be called. This restriction applies only to handlers in the same script in which you are using the `tell` command.

This statement causes the Control Panel window to instruct the Simulation movie to branch to another frame:

```
tell window "Simulation" to go frame "Save"
```

## tellStreamStatus()

### Syntax

```
tellStreamStatus(onOrOffBoolean)
```

### Description

Function; turns the stream status handler on (TRUE) or off (FALSE).

The form `tellStreamStatus()` determines the status of the handler.

When the `streamStatusHandler` is TRUE, Internet streaming activity causes periodic calls to the movie script, triggering `streamStatusHandler`. The handler is executed, with Director automatically filling in the parameters with information regarding the progress of the downloads.

### Examples

This `on prepareMovie` handler turns the `on streamStatus` handler on when the movie starts:

```
on prepareMovie
  tellStreamStatus(TRUE)
end
```

This statement determines the status of the stream status handler:

```
on mouseDown
    put tellStreamStatus()
end
```

**See also**

on streamStatus

## tellTarget()

**Syntax**

```
sprite(whichSprite).tellTarget(" targetName")
sprite(whichSprite).endTellTarget()
```

**Definition**

Command; equivalent to the Flash `beginTellTarget` and `endTellTarget` methods. The `tellTarget()` command allows the user to set a target Timeline on which subsequent sprite commands will act. When the target is set to a Flash movie clip or a level containing a loaded Flash movie, certain commands act on the targeted components, rather than on the main Timeline. To switch focus back to the main Timeline, call `endTellTarget()`.

The only valid argument for `tellTarget` is the target name. There is no valid argument for `endTellTarget`.

The Flash sprite functions that are affected by `tellTarget` are `stop`, `play`, `getProperty`, `setProperty`, `gotoFrame`, `call(frame)`, and `find(label)`. In addition, the `sprite` property `frame` (which returns the current frame) is affected by `tellTarget`.

**Examples**

This command sets the movie clip as the target:

```
sprite(1).tellTarget("\myMovieClip")
```

This command stops the movie clip:

```
sprite(1).stop()
```

This command causes the movie clip to play:

```
sprite(1).play()
```

This command switches the focus back to the main Timeline:

```
sprite(1).endTellTarget()
```

This command stops the main movie:

```
sprite(1).stop()
```

## tension

### Syntax

```
member(whichCastmember).model(whichModel).sds.tension
```

### Description

3D subdivision surface property; allows you to get or set a floating-point percentage between 0.0 and 100.0 that controls how tightly the newly generated surface matches the original surface. The higher this value, the more tightly the subdivided surface matches the original surface. The default is 65.0.

### Example

The following statement sets the `tension` property of the `sds` modifier of the model `Baby` to 35. If the `sds` modifier's `error` setting is low and its `depth` setting is high, this statement will have a very pronounced effect on `Baby`'s geometry.

```
member("scene").model("Baby").sds.tension = 35
```

### See also

`sds` (`modifier`), `error`, `depth` (3D)

## text

### Syntax

```
member(whichCastMember).text  
the text of member whichCastMember
```

### Description

Text cast member property; determines the character string in the field cast member specified by *whichCastMember*.

The `text` cast member property is useful for displaying messages and recording what the user types.

This property can be tested and set.

When you use Lingo to change the entire text of a cast member you remove any special formatting you have applied to individual words or lines. Altering the `text` cast member property reapplies global formatting. To change particular portions of the text, refer to lines, words, or items in the text.

When the movie plays back as an applet, this property's value is "" (an empty string) for a field cast member whose text has not yet streamed in.

To see an example of `text` used in a completed movie, see the `Forms and Post`, and `Text` movies in the `Learning/Lingo Examples` folder inside the `Director` application folder.

### Examples

This statement places the phrase "Thank you." in the empty cast member `Response`:

```
if member("Response").text = EMPTY then  
    member("Response").text = "Thank You."
```

This statement sets the content of cast member Notice to “You have made the right decision!”

```
member("Notice").text = "You have made the right decision!"
```

**See also**

selEnd, selStart

## texture

**Syntax**

```
member(whichCastmember).texture(whichTexture)  
member(whichCastmember).texture[index]  
member(whichCastmember).shader(whichShader).texture  
member(whichCastmember).model(whichModel).shader.texture  
member(whichCastmember).model(whichModel).shaderList.texture  
member(whichCastmember).model(whichModel).shaderList[index].texture  
member(whichCastmember).modelResource(whichParticleSystemModel)\  
Resource).texture
```

**Description**

3D element and shader property; an image object used by a shader to define the appearance of the surface of a model. The image is wrapped onto the geometry of the model by the shader.

The visible component of a shader is created with up to eight layers of textures. These eight texture layers are either created from bitmap cast members or image objects within Director or imported with models from 3D modeling programs.

Create and delete textures with the `newTexture()` and `deleteTexture()` commands.

Textures are stored in the texture palette of the 3D cast member. They can be referenced by name (*whichTexture*) or palette index (*textureIndex*). A texture can be used by any number of shaders. Changes to a texture will appear in all shaders which use that texture.

There are three types of textures:

`#fromCastmember`; the texture is created from a bitmap cast member using the `newTexture()` command.

`#fromImageObject`; the texture is created from a lingo image object using the `newTexture()` command.

`#importedFromFile`; the texture is imported with a model from a 3D modeling program.

For a complete list of texture properties, see 3D Lingo by Feature.

The texture of a particle system is a property of the model resource, whose type is `#particle`.

**Example**

This statement sets the texture property of the shader named WallSurface to the texture named BluePaint:

```
member("Room").shader("WallSurface").texture = \  
member("Room").texture("BluePaint")
```

**See also**

newTexture, deleteTexture

## textureCoordinateList

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).  
textureCoordinateList  
modelResourceObjectReference.textureCoordinateList
```

### Description

3D property; when used with a model resource whose type is #mesh, or with a meshDeform modifier attached to a model, this property allows you to get or set the textureCoordinateList property of the model resource.

The textureCoordinateList property is a list of sublists identifying locations in an image that are to be used when texture mapping a triangle. Each sublist consists of two values indicating a location in a texture map. The values must be between 0.0 and 1.0 so that they can be scaled to arbitrarily sized texture maps. The default is an empty list.

Manipulate `modelResource.face[index].textureCoordinates` or `model.meshdeform.mesh[index].face[index].textureCoordinates` to change the mapping between textureCoordinates and the corners of a mesh face.

### Example

```
put member(5,2).modelResource("mesh square").\  
    textureCoordinateList  
--[ [0.1, 0.1], [0.2, 0.1], [0.3, 0.1], [0.1, 0.2], [0.2, 0.2], \  
    [0.3, 0.2], [0.1, 0.3], [0.2, 0.3], [0.3, 0.3] ]
```

### See also

face, texture, meshDeform (modifier)

## textureCoordinates

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\  
    face[faceIndex].textureCoordinates  
modelResourceObject.face[faceIndex].textureCoordinates
```

### Description

3D property; identifies which elements in the textureCoordinateList to use for the *faceIndex*'d face. This property must be a list of three integers specifying indices in the textureCoordinateList, corresponding to the textureCoordinates to use for each corner of the mesh's face.

### See also

face, textureCoordinateList

## textureLayer

### Syntax

```
member(whichCastmember).model(whichModel).meshDeform.mesh[index].\  
    textureLayer.count  
member(whichCastmember).model(whichModel).meshdeform.mesh[index].\  
    texturelayer.add()  
member(whichCastmember).model(whichModel).meshdeform.mesh[index].\  
    texturelayer[index].textureCoordinateList.
```

### Description

3D `meshdeform` modifier properties; using these properties you can get and set information about the texture layers of a specified mesh.

You can have up to eight texture layers for a shader, each layer can contain only one texture, but the same texture can be specified for more than one layer. Texture layers are layers of textures used by shaders.

Use the following properties to access and manipulate texture layers:

`meshdeform.mesh[index].texturelayer.count` returns the number of texture layers for the specified mesh.

`model.meshdeform.mesh[index].texturelayer.add()` adds an empty texture layer to the specified mesh.

`model.meshdeform.mesh[index].texturelayer[index].texturecoordinatelist` allows you to get a list of `textureCoordinates` for a particular layer of the specified mesh. You can also use this property to copy texture coordinates between texture layers as follows:

```
model.meshdeform.texturelayer[a].texturecoordinatelist = \
    model.meshdeform.texturelayer[b].texturecoordinatelist
```

### See also

`meshDeform` (modifier), `mesh` (property), `textureCoordinateList`, `add` (3D texture), `count`, `texture`, `textureModelList`

## textureList

### Syntax

```
member(whichMember).model(whichModel).shader(whichShader).textureList
member(whichMember).model(whichModel).shader(whichShader).textureList[index]
```

### Description

3D shader property; determines the list of textures applied to the shader. A shader can have up to 8 layers of textures. When tested, this property returns a linear list of texture objects. When set without specifying an index, this property specifies the texture object to be applied to all layers. Setting the `textureList` property to `VOID` disables texturing for all layers. The default value is `VOID`.

To test or set the texture object for a specific texture layer, include an index value.

### Example

This statement sets the 3rd texture layer of the shader named “WallSurface” to the texture named “BluePaint” in the cast member named “Room”:

```
member(3).model("Car").shader("WallSurface").textureList[3] = \
    member("Room").texture("BluePaint")
```

### See also

`textureModelList`

## textureMember

### Syntax

`member(whichCastmember).textureMember`

### Description

3D cast member property; indicates the name of the bitmap cast member used as the source of the default texture for the 3D cast member.

The 3D cast member's `textureType` property must be set to `#member` for the `textureMember` property to be effective.

### Example

The following statement sets the `textureMember` property of the cast member named `YardScene` to `"Fence"`. If the `textureType` property of `YardScene` is set to `#member`, the cast member named `Fence` will be the source bitmap for the default texture in `YardScene`.

```
member("YardScene").textureMember = "Fence"
```

### See also

`textureType`

## textureMode

### Syntax

```
member(whichCastmember).shader(whichShader).textureMode  
member(whichCastmember).model(whichModel).shader.textureMode  
member(whichCastmember).model(whichModel).shaderList[[index]].\  
    textureMode
```

### Description

3D `#standard` shader property; specifies how the first texture layer is mapped onto the surface of the model. Use the `textureModeList` property to specify textures for layers other than the first layer. This property is ignored if the `#toon` modifier is applied to the model resource.

The possible values of this property are `#none`, `#wrapPlanar`, `#wrapCylindrical`, `#wrapSpherical`, `#reflection`, `#diffuseLight`, and `#specularLight`. For descriptions of these terms, see `textureModeList`.

### Example

This statement sets the value of the `textureMode` property of the first texture layer of the shader of the model named `Ball` to `#wrapSpherical`:

```
member("scene").model("Ball").shader.textureMode = #wrapSpherical
```

### See also

`textureModeList`



# textureModelList

## Syntax

```
member(whichCastmember).shader(whichShader).textureModelList  
member(whichCastmember).shader(whichShader).  
textureModelList[textureLayerIndex]  
member(whichCastmember).model(whichModel).shader.textureModelList  
member(whichCastmember).model(whichModel).shader.  
textureModelList[textureLayerIndex]
```

## Description

3D standard shader property; allows you to change how a textureLayer is mapped onto the surface of a model. This property is ignored if the #toon modifier is applied to the model resource. Possible values are as follows:

- #none uses the texture coordinate values originally defined for the model resource. This setting disables wrapTransform and wrapTransformList [textureLayerIndex].
- #wrapPlanar wraps the texture on the model surface as though it were being projected from an overhead projector. The shader's wrapTransformList [textureLayerIndex] is applied to the mapping space before the texture coordinates are generated in model space. With an identity wrapTransformList [textureLayerIndex] (the default), the planar mapping is oriented such that the texture is extruded along the Z axis with the texture's up direction along the Y axis.
- #wrapCylindrical wraps the texture around the surface as though the surface were placed in the middle of the texture and then the texture were rolled around the surface to form a cylinder. The wrapTransformList [textureLayerIndex] is applied to the mapping space before the texture coordinates are generated in model space. With an identity wrapTransformList [textureLayerIndex] (the default), the cylindrical mapping is oriented such that the texture is wrapped from the -Y axis, starting at the left edge of the texture, toward the +X axis, around the Z axis. The up direction of the texture is toward the +Z axis.
- #wrapSpherical wraps the texture around the surface as though the surface were placed in the middle of the texture and then all four corners of the texture were pulled up to meet at the top. The wrapTransformList [textureLayerIndex] is applied to the mapping space before the texture coordinates are generated in model space. With an identity wrapTransformList [textureLayerIndex], the spherical mapping is located at the model space origin and oriented such that the texture is wrapped from the -Y axis, starting at the left edge of the texture, toward the +X axis, around the Z axis. The up direction of the texture is toward the +Z axis.
- #reflection is similar to #wrapSpherical except that the new texture coordinates are continuously reprojected onto the surface from a fixed orientation. When the model rotates, the texture coordinates will not rotate with it. Simulates light reflected on an object by its environment. This setting disables wrapTransform.
- #diffuseLight generates diffuse light mapping texture coordinate values, one per vertex, and stores the results in the referenced mesh. This setting disables wrapTransform.
- #specularLight generates specular light mapping texture coordinate values, one per vertex, and stores the results in the referenced mesh. This setting disables wrapTransform.

### Example

In this example, a shader is set up to simulate a reflective garden ball. The shader's first textureLayer is set to a spherical mapping and the third textureLayer is set to use a #refection style mapping. The shader's textureList[3] entry will appear to reflected from the environment onto all models which use this shader.

```
member("scene").shader("GardenBall").textureList[1] = \
    member("scene").texture("FlatShinyBall")
member("scene").shader("GardenBall").textureModeList[1] = \
    #wrapSpherical
member("scene").shader("GardenBall").textureList[3] = \
    member("scene").texture("GardenEnvironment")
member("scene").shader("GardenBall").textureModeList[3] = \
    #reflection
```

### See also

textureTransformList, wrapTransform

## textureRenderFormat

### Syntax

getRenderServices().textureRenderFormat

### Description

3D `renderServices` property; allows you to get or set the default bit format used by all textures in all 3d cast members. Use a texture's `texture.RenderFormat` property to override this setting for specific textures only. Smaller sized bit formats (i.e 16 bit variants such as #rgba5551) use less hardware accelerator video ram, allowing you to make use of more textures before being forced to switch to software rendering. Larger sized bit formats (i.e. 32 bit variants such as #rgba8888) generally look better. In order to use alpha transparency in a texture, the last bit must be nonzero. In order to get smooth transparency gradations the alpha channel must have more than 1 bit of precision.

Each pixel formats has four digits, with each digit indicating the degree of precision for red, green, blue, and alpha. The value you choose determines the accuracy of the color fidelity (precision of the alpha channel) and the amount of memory used by the hardware texture buffer. You can choose a value that improves color fidelity or a value that allows you to fit more textures on the card. You can fit twice as many 16-bit textures as 32-bit textures in the same space. If a movie uses more textures than fit on a card at a the same time, Director switches to #software rendering.

You can specify any of the following values for `textureRenderFormat`:

- #rgba8888: 32 bit color mode with 8 bits each for red, green, blue, and alpha
- #rgba8880: same as above, with no alpha value
- #rgba5650: 16-bit color mode with no alpha; 5 bits for red, 6 for green, 5 for blue
- #rgba5550: 16-bit color mode with no alpha; 5 bits each for red, green, and blue, with no alpha measure
- #rgba5551: 16-bit color mode with 5 bits each for red, green, and blue; 1 bit for alpha
- #rgba4444: 16-bit color mode with 4 bits each for red, green, blue, and alpha

The default value is #rgba5551.

### Example

The following statement sets the global `textureRenderFormat` for the 3D member to `#rgba8888`. Each texture in this movie will be rendered in 32-bit color unless its `texture.renderFormat` property is set to a value other than `#default`.

```
getRendererServices().textureRenderFormat = #rgba8888
```

### See also

`renderer`, `preferred3DRenderer`, `renderFormat`, `getRendererServices()`

## textureRepeat

### Syntax

```
member(whichCastmember).shader(whichShader).textureRepeat  
member(whichCastmember).model(whichModel).shader.textureRepeat  
member(whichCastmember).model(whichModel).shaderList[[index]].\  
    textureRepeat
```

### Description

3D `#standard` shader property; controls the texture clamping behavior of the first texture layer of the shader. Use the `textureRepeatList` property to control this property for texture layers other than the first layer.

When `textureRepeat` is set to `TRUE` and the value of the x and/or y components of `shaderReference.textureTransform.scale` is less than 1, the texture is tiled (repeated) across the surface of the model.

When `textureRepeat` is set to `FALSE`, the texture will not tile. If the value of the x and/or y components of `shaderReference.textureTransform.scale` is less than 1, any area of the model not covered by the texture will be black. If the value of the x and/or y components of `shaderReference.textureTransform.scale` is greater than 1, the texture is cropped as it extends past the texture coordinate range.

The default value of this property is `TRUE`. This property is always `TRUE` when using the `#software` renderer.

### Example

The following statement sets the `textureRepeat` property of the first shader used by the model named `gbCyl3` to `TRUE`. The first texture in that shader will tile if the value of the x or y component of its `textureTransform` or `textureTransformList` property is less than 1.

```
member("scene").model("gbCyl3").shader.textureRepeat = TRUE
```

### See also

`textureTransform`, `textureTransformList`

# textureRepeatList

## Syntax

```
shaderReference.textureRepeatList[textureLayerIndex]
member(whichCastmember).shader(whichShader).textureRepeatList\
[textureLayerIndex]
member(whichCastmember).shader[shaderListIndex].textureRepeatList\
[textureLayerIndex]
member(whichCastmember).model(whichModel).shader.textureRepeatList\
[textureLayerIndex]
member(whichCastmember).model(whichModel).shaderList\
[shaderListIndex].textureRepeatList[textureLayerIndex]
```

## Description

3D standard shader property; allows you to get or set the texture clamping behavior of any texture layer. When TRUE, the default, the texture in textureLayerIndex can be tiled (repeated) several times across model surfaces. This can be accomplished by setting

shaderReference.textureTransform

[textureLayerIndex].scale to be less than 1 in x or y. When this value is set to FALSE, the texture will apply to a smaller portion of model surfaces, rather than tile across those surfaces, when the shaderReference.textureTransform

[textureLayerIndex].scale is less than 1 in x or y. Think of it as shrinking the source image within the frame of the original image and filling in black around the gap. Similarly, if

shaderReference.textureTransform[texture

LayerIndex].scale is set to be greater than 1 in x or y, the image will be cropped as the border of the texture is extended past the texture coordinate range.

## Example

The following code will textureMap a sphere entirely with a granite texture repeated 4 times across the surface, and a logo image which covers just 1/4 of the surface.

```
m = member(2).model("mySphere")
f = member(2).newTexture("granite", #fromCastmember, \
    member("granite"))
g = member(2).newTexture("logo", #fromCastmember, member("logo"))
s = member(2).newShader("s", #standard)
s.textureList[1] = g
s.textureList[2] = f
s.textureRepeatList[2] = false
s.textureRepeatList[1] = true
s.textureTransformList[1].scale(0.5,0.5,1.0)
s.textureTransformList[2].scale(0.5,0.5,1.0)
s.textureModeList[2] = #wrapPlanar
s.blendFunctionList[2] = #add
m.shaderList = s
```

# textureTransform

## Syntax

```
member(whichCastmember).shader(whichShader).textureTransform
member(whichCastmember).model(whichModel).shader.textureTransform
member(whichCastmember).model(whichModel).shaderList[[index]].\
    textureTransform
```

## Description

3D #standard shader property; provides access to a transform which modifies the texture coordinate mapping of the first texture layer of the shader. Manipulate this transform to tile, rotate, or translate the texture before applying it to the surface of the model. The texture itself remains unaffected; the transform merely modifies how the shader applies the texture. The textureTransform property is applied to all texture coordinates regardless of the textureMode property setting. This is the last modification of the texture coordinates before they are sent to the renderer. The textureTransform property is a matrix that operates on the texture in textureImage space. TextureImage space is defined to exist only on the X,Y plane.

To tile the image twice along its horizontal axis, use shaderReference.textureTransform.scale(0.5, 1.0, 1.0). Scaling on the Z axis is ignored.

To offset the image by point(xOffset,yOffset), use shaderReference.textureTransform.translate(xOffset,yOffset,0.0). Translating by integers when the shader's textureRepeat property is TRUE will have no effect, because the width and height of the texture will be valued between 0.0 and 1.0 in that case.

To apply a rotation to a texture layer, use shaderReference.textureTransform.rotate(0,0,angle). Rotations around the Z axis are rotated around the (0,0) 2d image point, which maps to the upper left corner of the texture. Rotations about the X and Y axes are ignored.

Just as with a model's transform, textureTransform modifications are layerable. To rotate the texture about a point(xOffset,yOffset) instead of point(0,0), first translate to point(0 - xOffset, 0 - yOffset), then rotate, then translate to point(xOffset, yOffset). The textureTransform is similar to the shader's wrapTransform property with the following exceptions. It is applied in 2d image space rather than 3d world space. As a result, only rotations about the Z axis and translations and scales on X and Y axes are effective. The transform is applied regardless of the shaderReference.textureMode setting. The wrapTransform, by comparison, is only effective when the textureMode is #wrapPlanar, #wrapCylindrical, or #wrapSpherical.

## Examples

This statement shows the textureTransform of the first texture in the first shader used by the model gbCyl3:

```
put member("Scene").model("gbCyl3").shader.textureTransform
-- transform(1.0000, 0.0000, 0.0000,0.0000, 0.0000, 1.0000, \
    0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 0.0000, 0.0000, \
    0.0000, 1.0000)
```

The following statement halves the height and width of the first texture used by the shader named gbCyl3. If the textureRepeat property of gbCyl3 is set to TRUE, four copies of the texture will be tiled across the shader.

```
member("Scene").shader("gbCyl3").textureTransform.scale = \
    vector(0.5, 0.5, 1)
```

This statement rotates the first texture used by the shader gbCyl3 by 90° from vector(0, 0, 0):

```
member("Scene").shader("gbCyl3").textureTransform.rotation = \
vector(0, 0, 90)
```

## textureTransformList

### Syntax

```
shaderReference .textureTransformList[textureLayerIndex]
member(whichCastmember).shader(ShaderName).textureTransformList\
[textureLayerIndex]
member(whichCastmember).shader[shaderListIndex].texture\
TransformList[textureLayerIndex]
member(whichCastmember).model(modelName).shader.texture\
TransformList[textureLayerIndex]
member(whichCastmember).model(modelName).shaderList\
[shaderListIndex]. textureTransformList[textureLayerIndex]
```

### Description

3D standard shader property; this property provides access to a transform which modifies the texture coordinate mapping of a texture layer. Manipulate this transform to tile, rotate, or translate a texture image before applying it to the surface of models. The texture itself remains unaffected, the transform merely modifies how the shader applies the texture.

To tile the image twice along its horizontal axis, use textureTransformList[whichTextureLayer].scale(0.5, 1.0, 1.0). Scales in Z will be ignored since images are 2D in nature. Care must be taken to avoid 0.0 scales (even in Z), as that will negate the effect of the entire texture.

To offset the image by point(xOffset,yOffset), use textureTransformList[whichTextureLayer].translate(xOffset,yOffset,0.0). Translating by integers when that texture layer's textureRepeat property is TRUE will have no effect, because the width and height of the texture will be valued between 0.0 and 1.0 in that case.

To apply a rotation to a texture layer, use textureTransformList[whichTextureLayer].rotate(0,0,angle). Rotations around the Z axis are rotated around the (0,0) 2D image point, which maps to the upper left corner of the texture. Rotations about X and Y will be ignored since images are 2D by nature.

Just as with a model's transform, textureTransform modifications are layerable. To rotate the image about a point(xOffset,yOffset) instead of point(0,0), first translate to point(0 - xOffset, 0 - yOffset), then rotate, then translate to point(xOffset, yOffset).

The textureTransformList is similar to the shaderwrapTransformList property with the following exceptions.

It is applied in 2D image space rather than 3D world space. As a result, only rotations in Z, and translations and scales in X and Y, are effective.

The transform is applied regardless of the shaderReference.textureModeList[index] setting. The wrapTransform, by comparison, is only effective when the textureMode is #wrapPlanar, #wrapCylindrical, or #wrapSpherical.

## Examples

This statement shows the textureTransform of the third texture in the first shader used by the model gbCyl3:

```
put member("scene").model("gbCyl3").shader.textureTransformList[3]
-- transform(1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000, \
    0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 0.0000, 0.0000, \
    0.0000, 1.0000)
```

The following statement halves the height and width of the fifth texture used by the shader gbCyl3. If the textureRepeatList[5] value of gbCyl3 is set to TRUE, four copies of the texture will be tiled across the shader.

```
member("scene").shader("gbCyl3").textureTransformList[5].scale = \
    vector(0.5, 0.5, 1)
```

This statement rotates the fourth texture used by the shader gbCyl3 by 90° from vector(0, 0, 0):

```
member("scene").shader("gbCyl3").textureTransformList[4].rotation \
    = vector(0, 0, 90)
```

These statements rotate the third texture used by the shader gbCyl3 by 90° around its center, assuming that textureList[3] is a 128x128 sized texture:

```
s = member("scene").shader("gbCyl3")
s.textureTransformList[3].translate(-64, -64, 0)
s.textureTransformList[3].rotate(0, 0, 90)
s.textureTransformList[3].translate(64, 64, 0)
```

## textureType

### Syntax

```
member(whichCastmember).textureType
```

### Description

3D texture property; allows you to get or set the texture type for the default texture. Possible values are as follows:

- `#none` specifies that there is no texture type.
- `#default` uses the texture from the original shader as the texture.
- `#member` uses the image from the specified cast member as the texture.

The default value for this property is `#default`. You must specify `#member` for this property in order to use the `textureMember` property.

### Example

The following statement sets the `textureType` property of the cast member Scene to `#member`.

```
member("Scene").textureType = #member
```

This makes it possible use a bitmap cast member as the source of the default texture by setting the `textureMember` property. The bitmap cast member is named "grass".

```
member("Scene").textureMember = "grass"
```

### See also

`textureMember`

## the

### Syntax

the *property*

### Description

Keyword; must precede many functions and all Lingo properties written in verbose syntax. This keyword also distinguishes the property or function from a variable or object name.

Earlier versions of Director required you to use the `the` keyword to express cast member and sprite properties. This syntax is still supported as alternate form.

Properties are globally available to handlers even if you don't declare them globally. Like global variables, Lingo system properties are available between different movies in the same presentation. Sprite properties change when a new movie is loaded.

## thumbnail

### Syntax

member(*whichMember*).thumbnail  
the thumbnail of member *whichMember*

### Description

Cast member property; contains the image used to preview a cast member in the Cast window. This image can be customized for any cast member.

This property can be tested and set only during authoring.

### Example

The following statement shows how to use a placeholder cast member to display another thumbnail on the Stage. The placeholder cast member is placed on the Stage, then the picture of that member is set to the thumbnail of member 10. This makes it possible to show a reduced image without having to scale or otherwise manipulate a graphic:

```
member("Placeholder").picture = member(10).thumbnail
```

### See also

picture (cast member property)

## ticks

### Syntax

the ticks

### Description

System property; returns the current time in ticks (1 tick = 1/60 of a second). Counting ticks begins from the time the computer is started.

### Example

This statement converts ticks to seconds and minutes by dividing the number of ticks by 60 twice and then sets the variable `minutesOn` to the result:

```
currentSeconds = the ticks/60  
currentMinutes = currentSeconds/60
```



**See also**

`time()`, `timer`, `milliseconds`

## tilt

**Syntax**

`tilt of sprite (whichQTVRSprite)`

**Description**

QuickTime VR sprite property; the current tilt, in degrees, of the QuickTime VR movie.

This property can be tested and set.

## time()

**Syntax**

`the time`  
`the short time`  
`the long time`  
`the abbreviated time`  
`the abbrev time`  
`the abbr time`

**Description**

Function; returns the current time in the system clock as a string in one of three formats: short, long, or abbreviated. If you don't specify a format, the default format is short. The abbreviated format can also be referred to as abbrev and abbr. In the United States, the short and abbreviated formats are the same.

**Example**

The following statements display the time in different formats in the Message window. Possible results appear below each statement.

```
put the short time
--"1:30 PM"
put the long time
--"1:30:24 PM"
put the abbreviated time
--"1:30 PM"
```

The three time formats vary, depending on the individual computer's time format. The preceding examples are for the United States.

**See also**

`date()` (system clock)

## time (timeout object property)

**Syntax**

`timeoutObject.time`

**Description**

Timeout object property; the system time, in milliseconds, when the next timeout event will be sent by the given *timeoutObject*.

Note that this is not the time until the next event, but the absolute time of the next timeout event.

#### Example

This handler determines the time remaining until the next timeout event will be sent by the timeout object `Update` by calculating the difference between its `time` property and the current value of the milliseconds and displaying the result in the field `Time Until`:

```
on prepareFrame
  msBeforeUpdate = timeout("Update").time - the milliseconds
  secondsBeforeUpdate = msBeforeUpdate / 1000
  minutesBeforeUpdate = secondsBeforeUpdate / 60
  member("Time Until").text = string(minutesBeforeUpdate) && "minutes before next \
  timeout"
end
```

#### See also

milliseconds, period, persistent, target, timeout(), timeoutHandler

## timeout()

#### Syntax

```
timeout("timeoutName")
```

#### Description

Function; returns the timeout object named *timeoutName*. Use `timeout("name").new` to add a new timeout object to the `timeoutList`. See `new()`.

#### Example

This handler deletes the timeout object named `Random Lightning`:

```
on exitFrame
  timeout("Random Lightning").forget()
end
```

#### See also

`forget()`, `new()`, `timeoutHandler`, `timeoutList`

## on timeOut

#### Syntax

```
on timeOut
  statement(s)
end
```

#### Description

System message and event handler; contains statements that run when the keyboard or mouse is not used for the time period specified in `timeOutLength`. Always place an `on timeOut` handler in a movie script.

To have a timeout produce the same response throughout a movie, use the `timeoutScript` to centrally control timeout behavior.

### Example

The following handler plays the movie *Attract Loop* after users do nothing for the time set in the `timeoutLength` property. It can be used to respond when users leave the computer.

```
on timeout
    play movie "Attract Loop"
end timeout
```

### See also

`timeoutScript`, `timeoutLength`

## timeoutHandler

### Syntax

*timeoutObject.timeoutHandler*

### Description

System property; represents the name of the handler that will receive timeout messages from the given *timeoutObject*. Its value is a symbol, such as `#timeExpiredHandler`. The `timeoutHandler` is always a handler within the timeout object's target object, or in a movie script if the timeout object has no target specified.

This property can be tested and set.

### Example

This statement displays the `timeoutHandler` of the timeout object *Quiz Timer* in the Message window:

```
put timeout("Quiz Timer").timeoutHandler
```

### See also

`target`, `timeout()`, `timeoutList`

## timeoutKeyDown

### Syntax

the `timeoutKeyDown`

### Description

System property; determines whether `keyDown` events set the `timeoutLapsed` property to 0 (TRUE, default) or not (FALSE). This property is useful for restarting the countdown for a timeout each time a key is pressed.

This property can be tested and set.

### Examples

This statement sets the variable `timing` to the value of the `timeoutKeyDown` property:

```
timing = timeoutKeyDown
```

This statement turns off the `timeoutKeyDown` property:

```
timeoutKeyDown = FALSE
```

### See also

`keyDownScript`

## timeoutLapsed

### Syntax

the timeoutLapsed

### Description

System property; indicates how many of ticks have elapsed since the last timeout. A timeout event occurs when the timeoutLapsed property reaches the time specified by the timeoutLength property.

The timeoutLapsed property can be tested and set.

### Example

The following statement sets the Countdown member field to the value of the timeoutLapsed property. Dividing timeoutLapsed by 60 converts the value to seconds.

```
member("Countdown").text = string(the timeoutLapsed / 60)
```

## timeoutLength

### Syntax

the timeoutLength

### Description

System property; determines how many ticks must elapse before a timeout event occurs. A timeout occurs when the timeoutLapsed property reaches the time specified by timeoutLength.

This property can be tested and set. The default value is 10,800 ticks or 3 minutes.

### Example

This statement sets timeoutLength to 10 seconds:

```
set the timeoutLength to 10 * 60
```

or

```
the timeoutLength = 10 * 60
```

## timeoutList

### Syntax

the timeoutList

### Description

System property; a linear list containing all currently active timeout objects. Use the forget() function to delete a timeout object.

Timeout objects are added to the timeoutList with the new() function.

### Example

This statement deletes the third timeout object from the timeout list:

```
the timeoutList[3].forget()
```

### See also

forget(), new(), timeout(), target, timeoutHandler

## timeoutMouse

### Syntax

the timeoutMouse

### Description

System property; determines whether mouseDown events reset the timeoutLapsed property to 0 (TRUE, default) or not (FALSE).

This property can be tested and set.

### Examples

This statement records the current setting of timeoutMouse by setting the variable named timing to the timeoutMouse:

```
timing = the timeoutMouse
```

The following statement sets the timeoutMouse property to FALSE. The result is that the timeoutLapsed property keeps its current value when the mouse button is pressed.

```
the timeoutMouse = FALSE
```

### See also

mouseDownScript, mouseUpScript

## timeoutPlay

### Syntax

the timeoutPlay

### Description

System property; determines whether the timeoutLapsed property will be set to TRUE when the movie is paused with the pause command. When TRUE, timeouts will occur when the movie is paused. When FALSE, timeouts will not occur when the movie is paused. The default value is FALSE.

This property can be tested and set.

### Example

This statement sets timeoutPlay to TRUE, which tells Lingo to reset the timeoutLapsed property to 0 after a movie is played:

```
set the timeoutPlay to TRUE
```

or

```
the timeoutPlay = TRUE
```

## timeoutScript

### Syntax

the timeoutScript

### Description

System property; determines the Lingo that Director executes as a primary event handler when a timeout occurs. The Lingo is written as a string, surrounded by quotation marks. The default value is EMPTY.

To define a primary event handler for timeouts, set `timeoutScript` to a string of the appropriate Lingo: either a simple statement or a calling statement for a handler. When the assigned event script is no longer appropriate, turn it off with the statement `set the timeoutScript to EMPTY`.

This property can be tested and set.

#### **Example**

This statement sets `timeoutScript` to a calling script for the handler `timeoutProcedure`:

```
the timeoutScript = "timeoutProcedure"
```

#### **See also**

on timeOut

## **timer**

#### **Syntax**

```
the timer
```

#### **Description**

System property; a free running timer that counts time in ticks (1 tick = 1/60 second). It has nothing to do with the `timeoutScript` property. It is used only for convenience in timing certain events. The `startTimer` command sets `timer` to 0.

The `timer` property is useful for determining the amount of time passed since the `startTimer` command was issued. For example, you can use `timer` to synchronize pictures with a soundtrack by inserting a delay that makes the movie wait until a certain amount of time has elapsed.

#### **Examples**

This behavior for a frame script creates a 2-second delay:

```
on beginSprite
    startTimer
end
on exitFrame
    if (the timer < 60 * 2) then go the frame
end
```

This statement sets the variable `startTicks` to the current timer value:

```
startTicks = the timer
```

#### **See also**

`lastClick()`, `lastEvent()`, `lastKey`, `lastRoll`, `startTimer`

## timeScale

### Syntax

`member(whichCastMember).timeScale`  
the timeScale of member *whichCastMember*

### Description

Cast member property; returns the time unit per second on which the digital video's frames are based. For example, a time unit in a QuickTime digital video is 1/600 of a second.

This property can be tested but not set.

### See also

`digitalVideoTimeScale`

## title

### Syntax

`window (whichWindow.title)`  
the title of window *whichWindow*

### Description

Window property; assigns a title to the window specified by *whichWindow*.

This property can be tested and set for windows other than the Stage.

### Example

This statement makes Action View the title of window X:

```
window("X").title = "Action View"
```

## titleVisible

### Syntax

`window (whichWindow.titleVisible)`  
the titleVisible of window *whichWindow*

### Description

Window property; specifies whether the window specified by *whichWindow* displays the window title in the window's title bar.

This property can be tested and set for windows other than the Stage.

### Example

This statement displays the title of the window named "Control\_panel" by setting the window's `titleVisible` property to `TRUE`:

```
window("Control_panel").titleVisible = TRUE
```

## to

The word `to` occurs in a number of Lingo constructs.

### See also

`char...of`, `item...of`, `line...of`, `word...of`, `repeat with`, `set...to`, `set...=`

## toon (modifier)

### Syntax

`member(whichCastmember).model(whichModel).toon.toonModifierProperty`

### Description

3D modifier; once you have added the `#toon` modifier to a model you can get and set the `#toon` modifier properties.

The toon modifier draws a model using only a handful of colors, and resulting in a cartoon style of rendering of the model's surface. When the `#toon` modifier is applied, the model's shader texture, `reflectionMap`, `diffuseLightMap`, `specularLightMap`, and `glossMap` properties are ignored.

When the `#toon` modifier is used in conjunction with the `#inker` modifier, the rendered effect is cumulative and varies depending on which modifier was first applied. The list of modifiers returned by the `modifier` property will list `#inker` or `#toon` (whichever was added first), but not both. The toon modifier can not be used in conjunction with the `#sds` modifier.

The `#toon` modifier has the following properties:

**Note:** For more detailed information about the following properties see the individual property entries.

- `style` allows you to get or set the style applied to color transitions. The following are the possible values:
  - `#toon` gives sharp transitions between available colors.
  - `#gradient` gives smooth transitions between available colors.
  - `#blackAndWhite` gives sharp transition between black and white.
- `colorSteps` allows you to get or set the number of different colors used for lighting calculations. When setting this value it is rounded down to nearest power of 2. Allowed values are 2, 4, 8, and 16. The default is 2.
- `shadowPercentage` allows you to get or set the percentage of the colors (`colorSteps`) defined for lighting used to render the shadowed portion of the model's surface. Possible values range from 0 to 100. The default is 50.
- `shadowStrength` allows you to get or set the level of darkness applied to the shadowed portion of the model's surface. Possible values are any non-negative floating-point number. The default value is 1.0.
- `highlightPercentage` allows you to get or set the percentage of the colors defined for lighting (`colorSteps`) used to render the highlighted portion of the model's surface. Possible values range from 0 to 100. The default is 50.
- `highlightStrength` allows you to get or set the level of brightness applied to the highlighted portion of the model's surface. Possible values are any non-negative floating-point number. The default value is 1.0.
- `lineColor` allows you to get or set the color of lines drawn by the inker. Possible values are any valid Lingo color object. The default value is `rgb (0, 0, 0)`, which is black.
- `creases` allows you to get or set whether lines are drawn in creases. This is a Boolean value; the default value is `True`.



- `creaseAngle`, if `creases` is set to `TRUE`, allows you to get or set how sensitive the line drawing function of the toon modifier is to the presence of creases.
- `boundary` allows you to get or set whether lines are drawn around the boundary of the surface. This is a Boolean value; the default value is `True`.
- `lineOffset` allows you to get or set where lines are drawn relative to the shaded surface and the camera. Negative lines move lines toward the camera. Positive values move lines away from the camera. Possible values are floating-point numbers from -100.0 to 100.0. The default value is -2.0.
- `useLineOffset` allows you to get or set whether `lineOffset` is on or off. This is a Boolean value; the default value is `False`.
- `silhouettes` allows you to get or set whether lines are drawn to define the edges along the border of a model, outlining its shape. This is a Boolean value; the default value is `True`.

**See also**

`addModifier`, `modifiers`, `sds (modifier)`, `inker (modifier)`

## top

**Syntax**

`sprite(whichSprite).top`  
the top of sprite *whichSprite*

**Description**

Sprite property; returns the top vertical coordinate of the bounding rectangle of the sprite specified by *whichSprite* as the number of pixels from the upper left corner of the Stage.

When the movie plays back as an applet, this property's value is relative to the upper left corner of the applet.

This property can be tested and set.

**Example**

This statement checks whether the top of sprite 3 is above the top of the Stage and calls the handler `offTopEdge` if it is:

```
if sprite(3).top < 0 then offTopEdge
```

**See also**

`bottom`, `height`, `locH`, `left`, `locV`, `right`, `width`

## top (3D)

**Syntax**

`modelResourceObjectReference.top`

**Description**

3D command; when used with a model resource whose type is `#box`, allows you to both get and set the `top` property of the model resource.

The `top` property determines whether the top of the box is sealed (`TRUE`) or open (`FALSE`). The default value is `TRUE`.

**Example**

This statement sets the `top` property of the model resource Gift box to `FALSE`, meaning the top of this box will be open:

```
member("3D World").modelResource("Gift box").top = FALSE
```

**See also**

back, bottom (3D), front

## topCap

**Syntax**

```
modelResourceObjectReference.topCap
```

**Description**

3D command; when used with a model resource whose type is `#cylinder`, allows you to both get and set the `topCap` property of the model resource.

The `topCap` property determines whether the top cap of the cylinder is sealed (`TRUE`) or open (`FALSE`). The default value for this property is `FALSE`.

**Example**

This statement sets the `topCap` property of the model resource Tube to `FALSE`, meaning the top of this cylinder will be open:

```
member("3D World").modelResource("Tube").topCap = FALSE
```

## topRadius

**Syntax**

```
modelResourceObjectReference.topRadius
```

**Description**

3D command; when used with a model resource whose type is `#cylinder`, allows you to both get and set the `topRadius` property of the model resource, as a floating-point value.

The `topRadius` property determines the radius of the top cap of the cylinder. This property must always be 0.0 or greater. The default value is 25.0. Setting `topRadius` to 0.0 produces a cone.

**Example**

The following statement sets the `topRadius` property of the model resource Tube to 0.0. If the bottom radius has a value greater than 0, models using Tube will be conical.

```
member("3D World").modelResource("Tube").topRadius = 0.0
```

## topSpacing

**Syntax**

```
chunkExpression.topSpacing
```

**Description**

Text cast member property; allows you to specify additional spacing applied to the top of each paragraph in the *chunkExpression* portion of the text cast member.

The value itself is an integer, with less than 0 indicating less spacing between paragraphs and greater than 0 indicating more spacing between paragraphs.

The default value is 0, which results in default spacing between paragraphs.

**Example**

This statement sets the `topSpacing` of the second paragraph in text cast member "myText" to 20:

```
member(1).paragraph[2].topSpacing = 20
```

**See also**

`bottomSpacing`

## trace

**Syntax**

```
the trace
```

**Description**

Movie property; specifies whether the movie's trace function is on (TRUE) or off (FALSE). When the trace function is on, the Message window displays each line of Lingo that is being executed.

This property can be tested and set.

**Example**

This statement turns the `trace` property on:

```
the trace = TRUE
```

**See also**

`traceLogFile`

## traceLoad

**Syntax**

```
the traceLoad
```

**Description**

Movie property; specifies the amount of information that is displayed about cast members as they load:

- 0—Displays no information.
- 1—Displays cast members' names.
- 2—Displays cast members' names, the number of the current frame, the movie name, and the file seek offset (the relative amount the drive had to move to load the media).

The default value for the `traceLoad` property is 0. This property can be tested and set.

**Example**

This statement causes the movie to display the names of cast members as they are loaded:

```
the traceLoad = 1
```

## traceLogFile

### Syntax

the traceLogFile

### Description

System property; specifies the name of the file in which the Message window display is written. You can close the file by setting the traceLogFile property to EMPTY (""). Any output that would appear in the Message window is written into this file. You can use this property for debugging when running a movie in a projector and when authoring.

### Examples

This statement instructs Lingo to write the contents of the Message window in the file "Messages.txt" in the same folder as the current movie:

```
the traceLogFile = the moviePath & "Messages.txt"
```

This statement closes the file that the Message window display is being written to:

```
the traceLogFile = ""
```

## trackCount (cast member property)

### Syntax

```
member(whichCastMember).trackCount()  
trackCount(member whichCastMember)
```

### Description

Digital video cast member property; returns the number of tracks in the specified digital video cast member.

This property can be tested but not set.

### Example

This statement determines the number of tracks in the digital video cast member Jazz Chronicle and displays the result in the Message window:

```
put member("Jazz Chronicle").trackCount()
```

## trackCount (sprite property)

### Syntax

```
sprite(whichDigitalVideoSprite).trackCount()  
trackCount(sprite whichSprite)
```

### Description

Digital video sprite property; returns the number of tracks in the specified digital video sprite.

This property can be tested but not set.

### Example

This statement determines the number of tracks in the digital video sprite assigned to channel 10 and displays the result in the Message window:

```
put sprite(10).trackCount()
```

## trackEnabled

### Syntax

```
sprite(whichDigitalVideoSprite).trackEnabled(whichTrack)  
trackEnabled(sprite whichSprite, whichTrack)
```

### Description

Digital video sprite property; indicates the status of the specified track of a digital video. This property is `TRUE` if the track is enabled and playing. This property is `FALSE` if the track is disabled and no longer playing or is not updating.

This property cannot be set. Use the `setTrackEnabled` property instead.

### Example

This statement checks whether track 2 of digital video sprite 1 is enabled:

```
put sprite(1).trackEnabled(2)
```

### See also

`setTrackEnabled`

## trackNextKeyTime

### Syntax

```
sprite(whichDigitalVideoSprite).trackNextKeyTime(whichTrack)  
trackNextKeyTime(sprite whichSprite, whichTrack)
```

### Description

Digital video sprite property; indicates the time of the keyframe that follows the current time in the specified digital video track.

This property can be tested but not set.

### Example

This statement determines the time of the keyframe that follows the current time in track 5 of the digital video assigned to sprite channel 15 and displays the result in the Message window:

```
put sprite(15).trackNextKeyTime(5)
```

## trackNextSampleTime

### Syntax

```
sprite(whichDigitalVideoSprite).trackNextSampleTime(whichTrack)  
trackNextSampleTime(sprite whichSprite, whichTrack)
```

### Description

Digital video sprite property; indicates the time of the next sample that follows the digital video's current time. This property is useful for locating text tracks in a digital video.

This property can be tested but not set.

### Example

This statement determines the time of the next sample that follows the current time in track 5 of the digital video assigned to sprite 15:

```
put sprite(15).trackNextSampleTime(5)
```

## trackPreviousKeyTime

### Syntax

```
sprite(whichDigitalVideoSprite).trackPreviousKeyTime(whichTrack)  
trackPreviousKeyTime(sprite whichSprite, whichTrack)
```

### Description

Digital video sprite property; reports the time of the keyframe that precedes the current time.

This property can be tested but not set.

### Example

This statement determines the time of the keyframe in track 5 that precedes the current time in the digital video sprite assigned to channel 15 and displays the result in the Message window:

```
put sprite(2).trackPreviousKeyTime(1)
```

## trackPreviousSampleTime

### Syntax

```
sprite(whichDigitalVideoSprite).trackPreviousSampleTime(whichTrack)  
trackPreviousSampleTime(sprite whichSprite, whichTrack)
```

### Description

Digital video sprite property; indicates the time of the sample preceding the digital video's current time. This property is useful for locating text tracks in a digital video.

This property can be tested but not set.

### Example

This statement determines the time of the sample in track 5 that precedes the current time in the digital video sprite assigned to channel 15 and displays the result in the Message window:

```
put sprite(15).trackPreviousSampleTime(5)
```

## trackStartTime (cast member property)

### Syntax

```
member(whichDigitalVideoCastmember).trackStartTime(whichTrack)  
trackStartTime(member whichCastMember, whichTrack)
```

### Description

Digital video cast member property; returns the start time of the specified track of the specified digital video cast member.

This property can be tested but not set.

### Example

This statement determines the start time of track 5 in the digital video cast member Jazz Chronicle and displays the result in the Message window:

```
put member("Jazz Chronicle").trackStartTime(5)
```

## trackStartTime (sprite property)

### Syntax

```
sprite(whichDigitalVideoSprite).trackStartTime(whichTrack)  
trackStartTime(sprite whichSprite, whichTrack)
```

### Description

Digital video sprite property; sets the starting time of a digital video movie in the specified sprite channel. The value of `trackStartTime` is measured in ticks.

This property can be tested but not set.

### Example

In the Message window, the following statement reports when track 5 in sprite channel 10 starts playing. The starting time is 120 ticks (2 seconds) into the track.

```
put sprite(10).trackStartTime(5)  
-- 120
```

### See also

`duration`, `movieRate`, `movieTime`

## trackStopTime (cast member property)

### Syntax

```
member(whichDigitalVideoCastmember).trackStopTime(whichTrack)  
trackStopTime(member whichCastMember, whichTrack)
```

### Description

Digital video cast member property; returns the stop time of the specified track of the specified digital video cast member. It can be tested but not set.

### Example

This statement determines the stop time of track 5 in the digital video cast member Jazz Chronicle and displays the result in the Message window:

```
put member("Jazz Chronicle").trackStopTime(5)
```

## trackStopTime (sprite property)

### Syntax

```
sprite(whichDigitalVideoSprite).trackStopTime(whichTrack)  
trackStopTime(sprite, whichSprite, whichTrack)
```

### Description

Digital video sprite property; returns the stop time of the specified track of the specified digital video sprite.

When a digital video movie is played, `trackStopTime` is when playback halts or loops if the `loop` property is turned on.

This property can be tested but not set.

**Example**

This statement determines the stop time of track 5 in the digital video assigned to sprite 6 and displays the result in the Message window:

```
put sprite(6).trackStopTime(5)
```

**See also**

movieRate, movieTime, trackStartTime (cast member property)

## trackText

**Syntax**

```
sprite(whichDigitalVideoSprite).trackText(whichTrack)  
trackText(sprite whichSprite, whichTrack)
```

**Description**

Digital video sprite property; provides the text that is in the specified track of the digital video at the current time. The result is a string value, which can be up to 32K characters long. This property applies to text tracks only.

This property can be tested but not set.

**Example**

This statement assigns the text in track 5 of the digital video assigned at the current time to sprite 20 to the field cast member Archives:

```
member("Archives").text = string(sprite(20).trackText(5))
```

## trackType (cast member property)

**Syntax**

```
member(whichDigitalVideoCastmember).trackType(whichTrack)  
trackType(member whichCastMember, whichTrack)
```

**Description**

Digital video cast member property; indicates which type of media is in the specified track of the specified cast member. Possible values are #video, #sound, #text, and #music.

This property can be tested but not set.

**Example**

The following handler checks whether track 5 of the digital video cast member Today's News is a text track and then runs the handler textFormat if it is:

```
on checkForText  
  if member("Today's News").trackType(5) = #text then textFormat  
end
```



## trackType (sprite property)

### Syntax

```
sprite(whichDigitalVideoSprite).trackType(whichTrack)  
trackType(sprite whichSprite, whichTrack)
```

### Description

Digital video sprite property; returns the type of media in the specified track of the specified sprite. Possible values are #video, #sound, #text, and #music.

This property can be tested but not set.

### Example

The following handler checks whether track 5 of the digital video sprite assigned to channel 10 is a text track and runs the handler `textFormat` if it is:

```
on checkForText  
    if sprite(10).trackType(5) = #text then textFormat  
end
```

## trails

### Syntax

```
sprite(whichSprite).trails  
the trails of sprite whichSprite
```

### Description

Sprite property; for the sprite specified by *whichSprite*, turns the trails ink effect on (1 or TRUE) or off (0 or FALSE). For the value set by Lingo to last beyond the current sprite, the sprite must be a puppet.

To erase trails, animate another sprite across these pixels or use a transition.

### Example

This statement turns on trails for sprite 7:

```
sprite(7).trails = 1
```

### See also

`directToStage`

## transform (command)

### Syntax

```
transform()  
transform(n1,n2,n3, ... ,n14,n15,n16)
```

### Description

3D command; this command creates a transform object. When this command is used without providing any parameters it creates a transform object equal to the identity transform. The identity transform has positional and rotational components of vector(0,0,0), and it has a scale component of vector(1,1,1). When this command is used while providing sixteen parameters in the form of *n1*,*n2*,*n3*, ... ,*n14*,*n15*,*n16* then this command creates a transform object using those 16 entries for the transform data.

## Examples

This statement creates an identity transform and stores it in the variable `tTransform`:

```
tTransform = transform()
```

This statement creates an identity transform by specifying all 16 of its elements, and it stores the new transform in the variable `tTransform`.

```
tTransform = transform(1.0000,0.0000,0.0000,0.0000, \
    0.0000,1.0000,0.0000,0.0000, 0.0000,0.0000,1.0000,0.0000, \
    0.0000,0.0000,0.0000,1.0000)
```

This statement creates a custom transform by specifying all 16 of its elements, and it stores the new transform in the variable `tTransform`. The transform created has a position property of vector(19.2884, 1.7649, 4.2426), a rotation property of vector(75.7007, 0.0000, -6.5847) and a scale property of vector(0.4904, 0.7297, 0.3493).

```
tTransform = transform(0.4872,-0.0562,0.0000,0.0000, \
    0.0795,0.1722,0.7071,0.0000, -0.0795,-0.1722,0.7071,0.0000, \
    19.2884,1.7649,4.2426,1.0000)
```

## See also

`transform (property)`, `preRotate`, `preTranslate()`, `preScale()`, `rotate`, `translate`, `scale (command)`

# transform (property)

## Syntax

```
member(whichCastmember).node(whichNode).transform
member(whichCastmember).node(whichNode).transform.transform\
    Property
member(whichCastmember).model(whichModel).bonesPlayer.\
    bone[boneID].transform
member(whichCastmember).model(whichModel).bonesPlayer.\
    bone[boneID].transform.transformProperty
```

## Description

3D property and command; allows you to get or set the transform associated with a particular node or a specific bone within a model using the `bonesPlayer` modifier. As a command, `transform` provides access to the various commands and properties of the transform object. A node can be a camera, group, light or model object.

For node objects, this property defaults to the identity transform. A node's transform defines the position, rotation and scale of the node relative to its parent object. If a node's parent is the World group object, then the `transform` property of the node has the same value as is returned by the `getWorldTransform()` command.

For bones within models using the `bonesPlayer` modifier, this property defaults in value to the transform assigned to the bone upon creation of the model file. The transform of a bone represents the bone's rotation relative to its parent bone and its position relative to its original joint position. The original joint position is determined upon creation of the model file.

You can use the following transform commands and properties with the `transform` property of node objects:

**Note:** This section only contains summaries, see the individual entries for more detailed information.

- `preScale` applies scaling before the current positional, rotational, and scale offsets held by the transform.
- `preTranslate` applies a translation before the current positional, rotational, and scale offsets held by the transform.
- `preRotate` applies a rotation before the current positional, rotational, and scale offsets held by the transform.
- `scale` (command) applies scaling after the current positional, rotational, and scale offsets held by the transform.
- `scale` (transform) allows you to get or set the degree of scaling of the transform.
- `translate` applies a translation after the current positional, rotational, and scale offsets held by the transform.
- `rotate` applies a rotation after the current positional, rotational, and scale offsets held by the transform.
- `position` (transform) allows you to get or set the positional offset of the transform.
- `rotation` (transform) allows you to get or set the rotational offset of the transform.

If you want to modify the `transform` property of a bone within a model, then you must store a copy of the original transform of the bone, modify the stored copy using the above commands and properties, then reset the bone's `transform` property so that it is equal to the modified transform. For example:

```
t = member("character").model("biped").bonesPlayer.bone[38].\
    transform.duplicate()
t.translate(25,0,-3)
member("character").model("biped").bonesPlayer.bone[38].\
    transform = t
```

### Example

This Lingo shows the transform of the model box, followed by the position and rotation properties of the transform:

```
put member("3d world").model("box").transform
-- transform(1.000000,0.000000,0.000000,0.000000, \
0.000000,1.000000,0.000000,0.000000, \
0.000000,0.000000,1.000000,0.000000, -\
94.144844,119.012825,0.000000,1.000000)
put member("3d world").model("box").transform.position
-- vector(-94.1448, 119.0128, 0.0000)
put member("3d world").model("box").transform.rotation
--vector(0.0000, 0.0000, 0.0000)
```

### See also

`interpolateTo()`, `scale` (transform), `rotation` (transform), `position` (transform),  
`bone`, `worldTransform`, `preRotate`, `preScale()`, `preTranslate()`

## transitionType

### Syntax

`member(whichCastMember).transitionType`  
the transitionType of member *whichCastMember*

### Description

Transition cast member property; determines a transition's type, which is specified as a number. The possible values are the same as the codes assigned to transitions for the `puppetTransition` command.

### Example

This statement sets the type of transition cast member 3 to 51, which is a pixel dissolve cast member:

```
member(3).transitionType = 51
```

## translate

### Syntax

```
member(whichCastmember).node(whichNode).translate(xIncrement, \
yIncrement, zIncrement [, relativeTo])
member(whichCastmember).node(whichNode).translate\
(translateVector [, relativeTo])
transform.translate(xIncrement, yIncrement, zIncrement \
[, relativeTo])
transform.translate(translateVector [, relativeTo])
```

### Description

3D command; applies a translation after the current positional, rotational, and scale offsets held by a referenced node's transform object or the directly referenced transform object. The translation must be specified as a set of three increments along the three corresponding axes. These increments may be specified explicitly in the form of *xIncrement*, *yIncrement*, and *zIncrement*, or by a *translateVector*, where the x component of the vector corresponds to the translation along the x axis, y about y axis, and z about z axis.

A node can be a camera, model, light or group object.

The optional *relativeTo* parameter determines which coordinate system's axes are used to apply the desired translational changes. The *relativeTo* parameter can have any of the following values:

- `#self` applies the increments relative to the node's local coordinate system (the x, y and z axes specified for the model during authoring). This value is used as the default if you use the `translate` command with a node reference and the *relativeTo* parameter is not specified.
- `#parent` applies the increments relative to the node's parent's coordinate system. This value is used as the default if you use the `translate` command with a transform reference and the *relativeTo* parameter is not specified.
- `#world` applies the increments relative to the world coordinate system. If a model's parent is the world, than this is equivalent to using `#parent`.
- `nodeReference` allows you to specify a node to base your translation upon, the command applies the translations relative to the coordinate system of the specified node.

## Examples

This example constructs a transform using the transform command, then it initializes the transform's position and orientation in space before assigning the transform to the model named mars. Finally this example displays the resulting position of the model.

```
t =transform()
t.transform.identity()
t.transform.rotate(0, 90, 0)
t.transform.translate(100, 0, 0)
gbModel = member("scene").model("mars")
gbModel.transform = t
put gbModel.transform.position
-- vector(100.0000, 0.0000, 0.0000)
```

This Lingo moves the model Bip 20 units along the x axis of its parent node:

```
put member("Scene").model("Bip").position
-- vector( -38.5000, 21.2500, 2.0000)
member("Scene").model("Bip").translate(20, 10, -0.5)
put member("Scene").model("Bip").position
-- vector( -18.5000, 31.2500, 1.5000)
```

## See also

transform (property), preTranslate(), scale (command), rotate

# translation

## Syntax

```
member(whichQuickTimeMember).translation
the translation of member whichQuickTimeMember
sprite(whichQuickTimeSprite).translation
the translation of sprite whichQuickTimeSprite
```

## Description

QuickTime cast member and sprite property; controls the offset of a QuickTime sprite's image within the sprite's bounding box.

This offset is expressed in relation to the sprite's default location as set by its `center` property. When `center` is set to `TRUE`, the sprite is offset relative to the center of the bounding rectangle; when `center` is set to `FALSE`, the sprite is offset relative to the upper left corner of the bounding rectangle.

The offset, specified in pixels as positive or negative integers, is set as a Director list: [*xTrans*, *yTrans*]. The *xTrans* parameter specifies the horizontal offset from the sprite's default location; the *yTrans* parameter specifies the vertical offset. The default setting is [0,0].

When the sprite's `crop` property is set to `TRUE`, the `translation` property can be used to mask portions of the QuickTime movie by moving them outside the bounding rectangle. When the `crop` property is set to `FALSE`, the `translation` property is ignored, and the sprite is always positioned at the upper left corner of the sprite's rectangle.

This property can be tested and set.

### Example

The following frame script assumes that the center property of the cast member of a 320-pixel-wide QuickTime sprite in channel 5 is set to `FALSE`, and its crop property is set to `TRUE`. It keeps the playhead in the current frame until the movie's horizontal translation point has moved to the right edge of the sprite, in 10-pixel increments. This has a wipe right effect, moving the sprite out of view to the right. When the sprite is out of view, the playhead continues to the next frame.

```
on exitFrame
    horizontalPosition = sprite(5).translation[1]
    if horizontalPosition < 320 then
        sprite(5).translation = sprite(5).translation + [10, 0]
        go the frame
    end if
end
```

## transparent

### Syntax

```
member(whichCastmember).shader(whichShader).transparent
member(whichCastmember).model(whichModel).shader.transparent
member(whichCastmember).model(whichModel).shaderList\
    [shaderListIndex].transparent
```

### Description

3D standard shader property; lets you get or set whether a model is blended using alpha values (`TRUE`) or is rendered as opaque (`FALSE`). The default value for this property is `TRUE` (alpha-blended).

The functionality of `shader.blend` is dependent upon this property.

All shaders have access to the `#standard` shader properties; in addition to these standard shader properties shaders of the types `#engraver`, `#newsprint`, and `#painter` have properties unique to their type. For more information, see the `newShader`.

### Example

The following statement causes the model Pluto to be rendered opaque. The setting of the blend property for the model's shader will have no effect.

```
member("scene").model("Pluto").shader.transparent = FALSE
```

### See also

`blendFactor`, `blend (3D)`

## triggerCallback

### Syntax

```
sprite(whichQTVRSprite).triggerCallback
triggerCallback of sprite whichQTVRSprite
```

### Description

QuickTime VR sprite property; contains the name of the handler that runs when the user clicks a hotspot in a QuickTime VR movie. The handler is sent two arguments: the `me` parameter and the ID of the hotspot that the user clicked.

The value that the handler returns determines how the movie processes the hotspot. If the handler returns `#continue`, the QuickTime VR sprite continues to process the hotspot normally. If the handler returns `#cancel`, the default behavior for the hotspot is canceled.

Set this property to 0 to clear the callback.

The QuickTime VR sprite receives the message first.

To avoid a decrease in performance, set the `triggerCallback` property only when necessary.

This property can be tested and set.

#### Example

The following statement sets the callback handler for a QuickTime VR sprite to the handler named `MyHotSpotCallback` when the playhead first enters the sprite span. Every time that hotspot is triggered, the `MyHotSpotCallback` handler is executed. When the playhead leaves the sprite span, the callback is canceled.

```
property pMySpriteNum, spriteNum

on beginSprite me
    pMySpriteNum = me.spriteNum
    sprite(pMySpriteNum).triggerCallback = #MyHotSpotCallback
end

on MyHotSpotCallback me, hotSpotID
    put "Hotspot" && hotSpotID && "was just triggered"
end

on endSprite me
    sprite(pMySpriteNum).triggerCallback = 0
end
```

## trimWhiteSpace (property)

#### Syntax

`member(whichMember).trimWhiteSpace`

#### Description

Cast member property. Determines whether the white pixels around the edge of a bitmap cast member are removed or left in place. This property is set when the member is imported. It can be changed in Lingo or in the Bitmap tab of the Property inspector.

## trimWhitespace() (function)

#### Syntax

`imageObject.trimWhitespace()`

#### Description

Function; removes any white pixels that lie outside the minimum bounding rectangle and returns the result in a new image object.

#### Example

This statement trims the white space from member `Flower` and returns the new, trimmed image object in the variable `trimmedImage`:

```
trimmedImage = member("flower").image.trimWhitespace()
```

**See also**

`crop()` (member command)

## TRUE

**Syntax**

TRUE

**Description**

Constant; represents the value of a logically true expression, such as  $2 < 3$ . It has a traditional numerical value of 1, but any nonzero integer evaluates to TRUE in a comparison.

**Example**

This statement turns on the `soundEnabled` property by setting it to TRUE:

```
the soundEnabled = TRUE
```

**See also**

FALSE, if

## tunnelDepth

**Syntax**

```
member(whichTextMember).tunnelDepth  
member(whichCastMember).modelResource(whichExtruderModel)\  
    Resource).tunnelDepth
```

**Description**

A 3D extruder model resource property, as well as a text cast member property. Using this property allows you to get or set the extrusion depth (the distance between the front and back faces) of a 3D text model resource. Possible values are floating point numbers between 1.0 and 100.0. The default value is 50.0.

It is recommended that you see `extrudeToMember` entry for more information about working with extruder model resources and text cast members.

**Example**

In this example, the cast member `logo` is a text cast member. The following statement sets the `tunnelDepth` of `logo` to 5. When `logo` is displayed in 3D mode, its letters will be very shallow.

```
member("logo").tunnelDepth = 5
```

In this example, the model resource of the model `Slogan` is extruded text. The following statement sets the `tunnelDepth` of `Slogan`'s model resource to 1000. `Slogan`'s letters will be extremely deep.

```
member("scene").model("Slogan").resource.tunnelDepth = 1000
```

**See also**

`extrude3D`



## tweened

### Syntax

```
sprite(whichSprite).tweened  
the tweened of sprite whichSprite
```

### Description

Sprite property; determines whether only the first frame in a new sprite is created as a keyframe (TRUE), or whether all frames in the new sprite are created as keyframes (FALSE).

This property does not affect playback and is useful only during Score recording.

This property can be tested and set.

### Example

When this statement is issued, newly created sprites in channel 25 have a keyframe only in the first frame of the sprite span:

```
sprite(25).tweened = 1
```

## tweenMode

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).tweenMode  
modelResourceObjectReference.tweenMode
```

### Description

3D particle property; allows you to get or set whether the color of a particle varies according to its speed or age. The `tweenMode` property can have the following values:

- `#velocity` alters the color of the particle between `colorRange.start` and `colorRange.end` based on the velocity of the particle.
- `#age` alters the color of the particle by linearly interpolating the color between `colorRange.start` and `colorRange.end` over the lifetime of the particle. This is the default setting for this property.

### Example

In this example, `ThermoSystem` is a model resource of the type `#particle`. This statement sets the `ThermoSystem`'s `tweenMode` to `#velocity`, so its slower particles will not reach the color specified by `colorRange.end`, while its faster particles will:

```
member(8,2).modelResource("thermoSystem").tweenMode = \  
    #velocitytype (light)
```

## type (cast member property)

### Syntax

```
member(whichCastMember).type  
the type of member whichCastMember  
member(whichCastMember, which castLib).type  
member whichCastMember of castLib whichCast.type  
the type of member whichCastMember of castLib whichCast
```

### Description

Cast member property; indicates the specified cast member's type. This property replaces the `castType` property used in previous versions of Director.

The `type` member property can be one of the following values:

---

<code>#animgif</code>	<code>#palette</code>
<code>#bitmap</code>	<code>#picture</code>
<code>#button</code>	<code>#QuickTimeMedia</code>
<code>#cursor</code>	<code>#script</code>
<code>#digitalVideo</code>	<code>#shape</code>
<code>#empty</code>	<code>#shockwave3D</code>
<code>#field</code>	<code>#sound</code>
<code>#filmLoop</code>	<code>#swa</code>
<code>#flash</code>	<code>#text</code> ( <code>#richText</code> is now obsolete)
<code>#font</code>	<code>#transition</code>
<code>#movie</code>	<code>#vectorShape</code>
<code>#ole</code>	

---

This list includes those types of cast members that are available in Director and the Xtra extensions that come with it. You can also define custom cast member types for custom cast members.

When a movie plays back as an applet, the `type` member property is valid only for cast member types that the player supports.

For movies created in Director 5 and 6, the `type` member property returns `#field` for field cast members and `#richText` for text cast members. However, field cast members originally created in Director 4 return `#text` for the member type, providing backward compatibility for movies that were created in Director 4.

This property can be tested but not set.

#### Example

The following handler checks whether the cast member Today's News is a field cast member and displays an alert if it is not:

```
on checkFormat
    if member("Today's News").type <> #field then alert \
        "Sorry, this cast member must be a field."
end
```

## type (light)

### Syntax

`member(whichCastmember).light(whichLight).type`

### Description

3D light property; the light type of the referenced light. This property's possible values are as follows:

- `#ambient` lights of this type cast their light evenly on all surfaces. The intensity of ambient lights is not affected by distance from the light source.
- `#directional` lights appear to shine in a particular direction, but are not as focused as lights of type `#spot`. The intensity of directional lights decreases with distance from the light source.
- `#point` lights shine in all directions from a specific location in the 3D world. The effect is similar to a bare light bulb hanging in a room. The intensity of point lights decreases with distance from the light source.
- `#spot` Lights of this type cast their light from a particular point and within the cone defined by the light's forward direction and `spotAngle` property. The intensity of spot lights declines with distance from the light source using the values defined in the light's `attenuation` property.

### Example

The following statement displays the type property of the light named `MainLight`:

```
put member("3D").motion("MainLight").type
-- #spot
```

### See also

`spotAngle`, `attenuation`

## type (model resource)

### Syntax

`member(whichCastmember).modelResource(whichModelResource).type`

### Description

3D model resource property; the resource type of the referenced model resource. This property's possible values are:

- `#box` indicates that this model resource is a primitive box resource created using the `newModelResource` command.
- `#cylinder` indicates that this model resource is a primitive cylinder resource created using the `newModelResource` command.
- `#extruder` indicates that this model resource is a primitive text extruder resource created using the `extrude3d` command.
- `#mesh` indicates that this model resource is a primitive mesh generator resource created using the `newMesh` command.
- `#particle` indicates that this model resource is a primitive particle system resource created using the `newModelResource` command.
- `#plane` indicates that this model resource is a primitive plane resource created using the `newModelResource` command.

- `#sphere` indicates that this model resource is a primitive sphere resource created using the `newModelResource` command.
- `#fromFile` indicates that this model resource was created external to Director and was loaded from an external file or a cast member.

#### Example

The following statement displays the `type` property of the model resource named `Helix`.

```
put member("helix models").modelResource("Helix").type
-- #fromFile
```

#### See also

`newModelResource`, `newMesh`, `extrude3D`

## type (motion)

#### Syntax

```
member(whichCastmember).motion(whichMotion).type
```

#### Description

3D motion property; the motion type of the referenced motion object. This property's possible values are:

- `#bonesPlayer` indicates that this motion is a bones based animation and it requires the use of the `#bonesPlayer` modifier for playback.
- `#keyFramePlayer` indicates that this motion is a keyframed animation and it requires the use of the `#keyFramePlayer` modifier for playback.
- `#none` indicates that this motion has no mapped movement and it is suitable for use by either the `#bonesPlayer` or the `#keyFramePlayer` modifier for playback. The default motion object found in every 3D cast member is of this type.

#### Examples

The following statement displays the `type` property of the motion named `Run`.

```
put member("scene").motion("Run").type
-- #bonesPlayer
```

The following statement displays the `type` property of the motion named `DefaultMotion`.

```
put member("scene").motion("DefaultMotion").type
-- #none
```

#### See also

`bonesPlayer (modifier)`, `keyframePlayer (modifier)`

## type (shader)

### Syntax

```
member(whichCastmember).shader(whichShader).type
```

### Description

3D shader property; the shader type of the referenced shader object. This property's possible values are:

- `#standard` indicates that this is a standard shader.
- `#painter` indicates that this is a painter shader.
- `#newsprint` indicates that this is a newsprint shader.
- `#engraver` indicates that this is an engraver shader.

### Examples

This statement shows that the shader used by the model named box2 is a painter shader:

```
put member("Scene").model("box2").shader.type  
-- #painter
```

### See also

`newShader`

## type (sprite property)

### Syntax

```
sprite(whichSprite).type  
the type of sprite whichSprite
```

### Description

Sprite property; clears sprite channels during Score recording by setting the `type` sprite property value for that channel to 0.

**Note:** Switch the member of a sprite only to another member of the same type to avoid changing the sprite's properties when the member type is switched.

This property can be tested and set.

### Example

This statement clears sprite channel 1 when issued during a Score recording session:

```
sprite(1).type = 0
```

## type (texture)

### Syntax

`member(whichCastmember).shader(whichShader).type`

### Description

3D texture property; the texture type of the referenced texture object. This property's possible values are:

- `#fromCastMember` indicates that this is texture was created from a Director cast member supporting the `image` property using the `newTexture` command.
- `#fromImageObject` indicates that this is texture was created from an image object using the `newTexture` command.
- `#importedFromFile` indicates that this texture was created external to Director and created upon file import or cast member loading.

### Example

This statement shows that the texture used by the shader for the model named Pluto was created from an image object:

```
put member("scene").model("Pluto").shader.texture.type  
-- #fromImageObject
```

### See also

`newTexture`

## union()

### Syntax

```
rect(1).union(rect(2))  
union (rect1, rect2)
```

### Description

Function; returns the smallest rectangle that encloses the two rectangles *rect1* and *rect2*.

### Example

This statement returns the rectangle that encloses the specified rectangles:

```
put union (rect (0, 0, 10, 10), rect (15, 15, 20, 20))  
-- rect (0, 0, 20, 20)
```

or

```
put rect(0, 0, 10, 10).union(rect(15, 15, 20, 20))  
--rect (0, 0, 20, 20)
```

### See also

map(), rect()

## unLoad

### Syntax

```
unLoad  
unLoad theFrameNum  
unLoad fromFrameNum, toFrameNum
```

### Description

Command; forces Director to clear the cast members used in a specified frame from memory. Director automatically unloads the least recently used cast members to accommodate *preLoad* commands or normal cast loading.

- When used without an argument, the *unLoad* command clears from memory the cast members in all the frames of a movie.
- When used with one argument, *theFrameNum*, the *unLoad* command clears from memory the cast members in that frame.
- When used with two arguments, *fromFrameNum* and *toFrameNum*, the *unLoad* command unloads all cast members in the range specified. You can specify a range of frames by frame numbers or frame labels.

### Examples

This statement clears the cast members used in frame 10 from memory:

```
unLoad 10
```

This statement clears the cast members used from the frame labeled *first* to the frame labeled *last*:

```
unLoad "first", "last"
```

### See also

*preLoad* (command), *preLoadMember*, *unLoadMember*, *purgePriority*

## unloadMember

### Syntax

```
unloadMember  
member(whichCastMember). unload()  
unloadMember member whichCastMember  
member(whichCastMember, whichCastLib). unload()  
unloadMember member whichCastMember of castLib whichCast  
member(firstCastmember). unload(lastCastMember)  
unloadMember member firstCastMember, lastCastMember
```

### Description

Command; forces Director to clear the specified cast members from memory. Director automatically unloads the least recently used cast members to accommodate `preLoad` commands or normal cast loading.

- When used without an argument, `unloadMember` clears from memory the cast members in all the frames of a movie.
- When used with the arguments *whichCastMember* and *whichCast*, the `unloadMember` command clears from memory the cast member name or number that you specify.
- When used with the arguments *firstCastMember* and *lastCastMember*, the `unloadMember` command unloads all cast members in the range specified.

When used in a new movie with no loaded cast members, this command returns an error.

Cast members that you have modified during authoring or by setting `picture`, `pasteClipBoardInto`, and so on, cannot be unloaded.

### Examples

This statement clears from memory the cast member `Screen1`:

```
unloadMember member "Screen1"
```

or

```
member("Screen1").unload()
```

This statement clears from memory all cast members from cast member 1 to cast member `Big Movie`:

```
unloadMember 1, member "Big Movie"
```

or

```
member(1).unload("Big Movie")
```

### See also

`preLoad` (command), `preLoadMember`, `purgePriority`

## unloadMovie

### Syntax

```
unloadMovie whichMovie
```

### Description

Command; removes the specified preloaded movie from memory. This command is useful in forcing movies to unload when memory is low.

You can use a URL as the file reference.



If the movie isn't already in RAM, the result is -1.

### Examples

This statement checks whether the largest contiguous block of free memory is less than 100K and unloads the movie Parsifal if it is:

```
if (the freeBlock < (100 * 1024)) then unLoadMovie "Parsifal"
```

This statement unloads the movie at <http://www.cbDemille.com/SunsetBlvd.dir>:

```
unLoadMovie "http://www.cbDemille.com/SunsetBlvd.dir"
```

## unregisterAllEvents

### Syntax

```
member(whichMember).unregisterAllEvents()
```

### Description

3D command; unregisters the referenced cast member for all event notifications. Therefore, all handlers that were previously registered to respond to events using the `registerForEvent` command will no longer be triggered when those events occur.

### Example

This statement unregisters the cast member named Scene for all event notifications:

```
member("Scene").unregisterAllEvents()
```

### See also

```
registerForEvent()
```

## update

### Syntax

```
member(whichCastmember).model(whichModel).update
```

### Description

3D command; causes animations on the model to update without rendering. Use this command to find the exact position of an animating model in Lingo.

## updateFrame

### Syntax

```
updateFrame
```

### Description

Command; during Score generation only, enters the changes to the current frame that have been made during Score recording and moves to the next frame. Any objects that were already in the frame when the update session started remain in the frame. You must issue an `updateFrame` command for each frame that you are updating.

### Example

When used in the following handler, the `updateFrame` command enters the changes that have been made to the current frame and moves to the next frame each time Lingo reaches the end of the repeat loop. The number of frames is determined by the argument `numberOfFrames`.

```
on animBall numberOfFrames
  beginRecording
    horizontal = 0
    vertical = 300
    repeat with i = 1 to numberOfFrames
      go to frame i
      sprite(20).memberNum = member("Ball").number
      sprite(20).locH = horizontal
      sprite(20).locV = vertical
      sprite(20).type = 1
      sprite(20).foreColor = 255
      horizontal = horizontal + 3
      vertical = vertical + 2
      updateFrame
    end repeat
  endRecording
end
```

### See also

`beginRecording`, `endRecording`, `scriptNum`, `tweened`

## updateLock

### Syntax

the `updateLock`

### Description

Movie property; determines whether the Stage is updated during Score recording (`FALSE`) or not (`TRUE`).

You can keep the Stage display constant during a Score recording session by setting `updateLock` to `TRUE` before Lingo updates the Score. If `updateLock` is `FALSE`, the Stage updates to show a new frame each time the frame is entered.

You can also use `updateLock` to prevent unintentional Score updating when leaving a frame, such as when you temporarily leave a frame to examine properties in another frame.

Although this property can be used to mask changes to a frame during run time, be aware that changes to field cast members appear immediately when the content is modified, unlike changes to location or members with other sprites, which are not updated until this property is turned off.

## updateMovieEnabled

### Syntax

the `updateMovieEnabled`

### Description

System property; specifies whether changes made to the current movie are automatically saved (`TRUE`) or not saved (`FALSE`, default) when the movie branches to another movie.

This property can be tested and set.

**Example**

This statement instructs Director to save changes to the current movie whenever the movie branches to another movie:

```
the updateMovieEnabled = TRUE
```

## updateStage

**Syntax**

```
updateStage
```

**Description**

Command; redraws the Stage immediately instead of only between frames.

The `updateStage` command redraws sprites, performs transitions, plays sounds, sends a `prepareFrame` message (affecting movie and behavior scripts), and sends a `stepFrame` message (which affects `actorList`).

**Example**

This handler changes the sprite's horizontal and vertical locations and redraws the Stage so that the sprite appears in the new location without having to wait for the playhead to move:

```
on moveRight whichSprite, howFar
    sprite(whichSprite).locH = sprite(whichSprite).locH + howFar
    updateStage
end moveRight
```

## URL

**Syntax**

```
member(whichCastMember).URL
the URL of member whichCastMember
```

**Description**

Cast member property; specifies the URL for Shockwave Audio (SWA) and Flash movie cast members.

For Flash movie members, this property is synonymous with the `pathName` member property.

The `URL` property can be tested and set. For SWA members, it can be set only when the SWA streaming cast member is stopped.

**Example**

This statement makes a file on an Internet server the URL for SWA cast member Benny Goodman:

```
on mouseDown
    member("Benny Goodman").URL = "http://audio.macromedia.com/samples/
    classic.swa"
end
```

## URLEncode

### Syntax

```
URLEncode(proplist_or_string {, serverOSSString} {, characterSet})
```

### Description

Function; returns the URL-encoded string for its first argument. Allows CGI parameters to be used in other commands. The same translation is done as for `postNetText` and `getNetText()` when they are given a property list.

Use the optional parameter *serverOSSString* to encode any return characters in *proplist\_or\_string*. The value defaults to "Unix" but may be set to "Win" or "Mac" and translates any carriage returns in the *proplist\_or\_string* argument into those used on the server. For most applications, this setting is unnecessary because line breaks are usually not used in form responses.

The optional parameter *characterSet* applies only if the user is running on a Shift-JIS (Japanese) system. Its possible settings are "JIS", "EUC", "ASCII", and "AUTO". Retrieved data is converted from Shift-JIS to the named character set. Returned data is handled exactly as by `getNetText()` (converted from the named character set to Shift-JIS). If you use "AUTO", the posted data from the local character set is not translated; the results sent back by the server are translated as they are for `getNetText()`. "ASCII" is the default if *characterSet* is omitted. "ASCII" provides no translation for posting or results.

### Example

In the following example, `URLEncode` supplies the URL-encoded string to a CGI query at the specified location.

```
URL = "http://aserver/cgi-bin/echoquery.cgi"  
gotonetpage URL & "?" & URLEncode( [#name: "Ken", #hobby: "What?"] )
```

### See also

`getNetText()`, `postNetText`

## useAlpha

### Syntax

```
member(whichCastMember).useAlpha  
imageObject.useAlpha
```

### Description

Bitmap cast member and image object property; for 32-bit cast members and image objects with alpha channel information, determines whether Director uses the alpha information when drawing the image onto the Stage (TRUE), or whether Director ignores the alpha information when drawing to the Stage (FALSE).

### Example

This toggles the alpha channel of cast member "foreground" on and off:

```
member("foreground").useAlpha=not member("foreground").useAlpha
```

## useDiffuseWithTexture

### Syntax

`member(whichCastmember).shader(whichShader).useDiffuseWithTexture`

### Description

3D standard shader property; allows you to get or set whether the diffuse color is used to modulate the texture (TRUE) or not (FALSE).

When set to TRUE, this property works in conjunction with the `blendFunction` and `blendConstant` properties: when `blendFunction` is set to `#blend`, the diffuse color is weighed with the texture color to determine the final color. For example, if `blendFunction` is set to `#blend`, and `blendConstant` is set to 100.0, the final color is the pure texture color. If we change `blendConstant` to 0.0, the final color is the diffuse color. If we change `blendConstant` to 10.0, the final color is 10% texture color, and 90% diffuse color.

The default value for this property is FALSE.

All shaders have access to the `#standard` shader properties; in addition to these standard shader properties shaders of the types `#engraver`, `#newsprint`, and `#painter` have properties unique to their type. For more information, see `newShader`.

### Example

In this example, the `shaderList` of the model `MysteryBox` contains six shaders. Each shader has a texture list which contains up to eight textures. The `diffuseColor` property of the cast member (`Level2`) is set to `rgb(255, 0, 0)`. The `blendFunction` property of all six shaders is set to `#blend`, and the `blendConstant` property of all six shaders is set to 80. This statement sets the `useDiffuseWithTexture` property of all shaders used by `MysteryBox` to TRUE. A little bit of red will be blended into the surface of the model. This property is affected by the settings of the `blendFunction`, `blendFunctionList`, `blendSource`, `blendSourceList`, `blendConstant`, and `blendConstantList` properties.

```
member("Level2").model("MysteryBox").shaderlist.useDiffuseWith\
Texture = TRUE
```

### See also

`blendFunction`, `blendConstant`

## useFastQuads

### Syntax

`the useFastQuads`

### Description

Global property; when set to TRUE, Director uses a faster, less precise method for calculating quad operations. Fast quads calculations are good for simple rotation and skew sprite effects. The slower, default quad calculation method available in Director provides more visually pleasing results when using quads for distortion and other arbitrary effects. Defaults to FALSE.

Simple sprite rotation and skew operations always use the fast quad calculation method, regardless of this setting. Setting the `useFastQuads` to TRUE will not result in an increase in the speed of these simple operations.

### Example

This statement tells Director to use its faster quad calculation code for all quad operations in the movie:

```
the useFastQuads = TRUE
```

### See also

quad

## useHypertextStyles

### Syntax

```
member(whichTextMember).useHypertextStyles
```

### Description

Text cast member property; controls the display of hypertext links in the text cast member.

When `useHypertextStyles` is `TRUE`, all links are automatically colored blue with underlines, and the pointer (cursor) changes to a pointing finger when it is over a link.

Setting this property to `FALSE` turns off the automatic formatting and pointer change.

### Example

This behavior toggles the formatting of hypertext on and off in text cast member “myText”:

```
on mouseUp
    member("myText").usehypertextStyles = not
    member("myText").usehypertextStyles
end
```

## useLineOffset

### Syntax

```
member(whichCastmember).model(whichModel).toon.useLineOffset
member(whichCastmember).model(whichModel).inker.useLineOffset
```

### Description

3D toon and inker modifier property; indicates whether the modifier’s `lineOffset` property is used by the modifier when it draws lines on the surface of the model.

The default value of this property is `FALSE`.

### Example

The following statement sets the `useLineOffset` property of the `toon` modifier for the model named `Teapot` to `FALSE`. The `toon` modifier’s `lineOffset` property will have no effect.

```
member("tp").model("Teapot").toon.useLineOffset = FALSE
```

### See also

lineOffset

## userData

### Syntax

```
member(whichCastmember).model(whichModel).userData  
member(whichCastmember).light(whichLight).userData  
member(whichCastmember).camera(whichCamera).userData  
member(whichCastmember).group(whichCamera).userData
```

### Description

3D property; returns the `userData` property list of a model, group, camera, or light. The default value of this property for an object that was created outside of Director is a list of all the properties that were assigned to the model's `userData` property in the 3D modeling tool. The default value of this property for objects created inside of Director is an empty property list [:], unless the object was created using any of the clone commands. If a cloning command was used to create the object then the new object's `userData` property defaults to a value equal to that of the original source object.

To modify the elements of this list you must use the `addProp` and `deleteProp` commands documented in the main Lingo Dictionary.

### Examples

This statement displays the `userData` property of the model named New Body:

```
put member("Car").model("New Body").userData  
-- [#driver: "Bob", #damage: 34]
```

This statement adds the property `#health` with the value 100 to the `userData` property list for the model named Player:

```
member("scene").model("Player").userData.addProp(#health,100)
```

## userName

### Syntax

```
the userName
```

### Description

Movie property; a string containing the user name entered when Director was installed.

This property is available in the authoring environment only. It could be used in a movie in a window (MIAW) tool that is personalized to show the user's information.

### Example

The following handler places the user's name and serial number in a display field when the window is opened. (A movie script in the MIAW is a good location for this handler.)

```
on prepareMovie  
  displayString = the userName  
  put RETURN&the organizationName after displayString  
  put RETURN&the serialNumber after displayString  
  member("User Info").text = displayString  
end
```

### See also

organizationName, serialNumber, window

## userName (RealMedia)

### Syntax

```
sprite(whichSprite).userName  
member(whichCastmember).userName  
sprite(whichSprite).userName = userName  
member(whichCastmember).userName = userName
```

### Description

RealMedia sprite and cast member property; allows you to set the user name required to access a protected RealMedia stream. For security reasons, you cannot use this property to retrieve a previously specified user name. If a user name has been set, the value of this property is the string "\*\*\*\*\*". The default value of this property is an empty string, which means no user name has been specified.

### Examples

The following examples show that the user name for the RealMedia stream in the cast member Real or sprite 2 has been set.

```
put sprite(2).userName  
-- "*****"  
  
put member("Real").userName  
-- "*****"
```

The following examples show that the user name for the RealMedia stream in the cast member Real or sprite 2 has never been set.

```
put sprite(2).userName  
-- ""  
  
put member("Real").userName  
-- ""
```

The following examples set the user name for the RealMedia stream in the cast member Real and sprite 2 to Marcelle.

```
member("Real").userName = "Marcelle"  
sprite(2).userName = "Marcelle"
```

### See also

password

## useTargetFrameRate

### Syntax

```
sprite(which3dSprite).useTargetFrameRate
```

### Description

3D sprite property; determines whether the `targetFrameRate` property of the sprite is enforced. If the `useTargetFrameRate` property is set to `TRUE`, the polygon count of the models in the sprite are reduced if necessary to achieve the specified frame rate.

### Example

These statements set the `targetFrameRate` property of sprite 3 to 45 and enforce the frame rate by setting the `useTargetFrameRate` property of the sprite to `TRUE`:

```
sprite(3).targetFrameRate = 45  
sprite(3).useTargetFrameRate = TRUE
```

### See also

targetFrameRate



# value()

## Syntax

`value(stringExpression)`

## Description

Function; returns the value of a string. The string can be any expression that Lingo can understand. When `value()` is called, Lingo parses through the *stringExpression* provided and returns its logical value.

Any Lingo expression that can be put in the Message window or set as the value of a variable can also be used with `value()`.

These two Lingo statements are equivalent:

```
put sprite(2).member.duration * 5
put value("sprite(2).member.duration * 5")
```

These two Lingo statements are also equivalent:

```
x = (the mouseH - 10) / (the mouseV + 10)
x = value("(the mouseH - 10) / (the mouseV + 10)")
```

Expressions that Lingo cannot parse will produce unexpected results, but will not produce Lingo errors. The result is the value of the initial portion of the expression up to the first syntax error found in the string.

The `value()` function can be useful for parsing expressions input into text fields by end-users, string expressions passed to Lingo by Xtra extensions, or any other expression you need to convert from a string to a Lingo value.

Keep in mind that there may be some situations where using `value()` with user input can be dangerous, such as when the user enters the name of a custom handler into the field. This will cause the handler to be executed when it is passed to `value()`.

Do not confuse the actions of the value function with the `integer()` and `float()` functions.

## Examples

This statement displays the numerical value of the string "the sqrt of" && "2.0":

```
put value("the sqrt of" && "2.0")
```

The result is 1.4142.

This statement displays the numerical value of the string "penny":

```
put value("penny")
```

The resulting display in the Message window is VOID, because the word *penny* has no numerical value.

You can convert a string that is formatted as a list into a true list by using this syntax:

```
myString = "[" & QUOTE & "cat" & QUOTE & ", " & QUOTE & "dog" & QUOTE & "]"
myList = value(myString)
put myList
-- ["cat", "dog"]
```

This allows a list to be placed in a field or text cast member and then extracted and easily reformatted as a list.

This statement parses the string "3 5" and returns the value of the portion of the string that Lingo understands:

```
put value("3 5")
-- 3
```

**See also**

`string()`, `integer()`, `float()`

## vector()

**Syntax**

```
vector (x, y, z)
```

**Description**

3D data type and function; a vector describes a point in 3D space according to the parameters *x*, *y*, and *z* which are the specific distances from the reference point along the *x*-axis, *y*-axis, and *z*-axis, respectively. If the vector is in world space, the reference point is the world origin, `vector(0, 0, 0)`. If the vector is in object space, the reference point is the object's position and orientation. This function returns a vector object.

Vector values can be operated upon by the `+`, `-`, `*` and `/` operators. See their individual definitions for more information.

**Examples**

This statement creates a vector and assigns it to the variable `MyVector`:

```
MyVector = vector(10.0, -5.0, 0.0)
```

This statement adds two vectors and assigns the resulting value to the variable `ThisVect`:

```
ThisVect = vector(1.0, 0.0, 0.0) + vector(0.0, -12.5, 2.0)
put ThisVect
-- vector(1.0000, -12.5000, 2.0000)
```

**See also**

`+` (addition) (3D), `-` (subtraction), `*` (multiplication), `/` (division) (3D)

## version

**Syntax**

```
version
```

**Description**

Keyword; system variable that contains the version string for Director. The same string appears in the Macintosh Finder's Info window.

**Example**

This statement displays the version of Director in the Message window:

```
put version
```

## vertex

### Syntax

`member(whichVectorShapeMember).vertex[whichVertexPosition]`

### Description

Chunk expression; enables direct access to parts of a vertex list of a vector shape cast member.

Use this chunk expression to avoid parsing different chunks of the vertex list. It's possible to both test and set values of the vertex list using this type of chunk expression.

### Examples

The following code shows how to determine the number of vertex points in a member:

```
put member("Archie").vertex.count
-- 2
```

To obtain the second vertex for the member, you can use code like this:

```
put member("Archie").vertex[2]
-- point(66.0000, -5.0000)
```

You can also set the value in a control handle:

```
member("Archie").vertex[2].handle1 = point(-63.0000, -16.0000)
```

### See also

`vertexList`

## vertexList

### Syntax

`member(whichVectorShapeMember).vertexList`

### Description

Cast member property; returns a linear list containing property lists, one for each vertex of a vector shape. The property list contains the location of the vertex and the control handle. There are no control handles if the location is (0,0).

Each vertex can have two control handles that determine the curve between this vertex and the adjacent vertices. In `vertexList`, the coordinates of the control handles for a vertex are kept relative to that vertex, rather than absolute in the coordinate system of the shape. If the first control handle of a vertex is located 10 pixels to the left of that vertex, its location is stored as (-10, 0). Thus, when the location of a vertex is changed with Lingo, the control handles move with the vertex and do not need to be updated (unless the user specifically wants to change the location or size of the handle).

When modifying this property, be aware that you must reset the list contents after changing any of the values. This is because when you set a variable to the value of the property, you are placing a copy of the list, not the list itself, in the variable. To effect a change, use code like this:

```
- Get the current property contents
currVertList = member(1).vertexList
-- Add 25 pixels to the horizontal and vertical positions of the first vertex
  in the list
currVertList[1].vertex = currVertList[1].vertex + point(25, 25)
-- Reset the actual property to the newly computed position
member(1).vertexList = currVertList
```

### Example

This statement displays the `vertexList` value for an arched line with two vertices:

```
put member("Archie").vertexList
-- [[/#vertex: point(-66.0000, 37.0000), #handle1: point(-70.0000, -36.0000),
   \
   #handle2: point(-62.0000, 110.0000)], [/#vertex: point(66.0000, -5.0000), \
   #handle1: point(121.0000, 56.0000), #handle2: point(11.0000, -66.0000)]]
```

### See also

`addVertex`, `count()`, `deleteVertex()`, `moveVertex()`, `moveVertexHandle()`, `originMode`, `vertex`

## vertexList (mesh generator)

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).vertexList
```

### Description

3D property; when used with a model resource whose type is `#mesh`, allows you to get or set the `vertexList` property for the model resource.

The `vertexList` is a linear list of each vertex used in the mesh. A single vertex may be shared by numerous faces of the mesh. You can specify a list of any size for this property, but it will store only the number of items specified when using the `newMesh()` command to create the `#mesh` model resource.

### Example

This statement sets the `vertexList` of the model resource named `Triangle`:

```
member("Shapes").modelResource("Triangle").vertexList = \
    [vector(0,0,0), vector(20,0,0), vector(20, 20, 0)]
```

### See also

`newMesh`, `face`, `vertices`

## vertexList (mesh deform)

### Syntax

```
member(whichCastmember).model(whichModel).meshDeform.mesh\
    [index].vertexList
```

### Description

3D property; when used with a model with the `#meshDeform` modifier attached, it allows you to get or set the `vertexList` property for the specified mesh within the referenced model.

The `vertexList` is a linear list of each vertex used in the specified mesh. A single vertex may be shared by numerous faces of the mesh.

If a model makes use of the `#sds` or `#lod` modifiers in addition to the `#meshDeform` modifier, then it is important to know that the value of this property will change under the influence of the `#sds` or `#lod` modifiers.

### Example

This statement displays the `#meshDeform` modifier's `vertexList` for the first mesh in the model named `Triangle`:

```
put member("Shapes").model("Triangle").meshDeform.mesh[1].vertexList
-- [vector(0,0,0), vector(20,0,0), vector(20, 20, 0)]
```

### See also

`face`, `vertices`, `mesh` (property)

## vertices

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).\
    face[faceIndex].vertices
```

### Description

3D face property; when used with a model resource whose type is `#mesh`, this property allows you to get or set which vertices from the resource's `vertexList` property to use for the mesh face specified by *faceIndex*.

This property is a linear list of three integers corresponding to the index positions of the three vertices, as found in the mesh's `vertexList` property, that comprise the specified face.

The vertices must be specified in the list using counterclockwise winding in order to achieve an outward pointing surface normal.

If you make changes to this property or use the `generateNormals()` command, you will need to call the `build()` command in order to rebuild the mesh.

### Example

This example displays the `vertexList` of the mesh model resource named `SimpleSquare`, then it displays the `vertices` property for the second face of that mesh:

```
put member("3D").modelResource("SimpleSquare").vertexList
-- [vector( 0.0000, 0.0000, 0.0000), vector( 0.0000, 5.0000, \
    0.0000), vector( 5.0000, 0.0000, 0.0000), vector( 5.0000, \
    5.0000, 0.0000)]
put member("3D").modelResource("SimpleSquare").face[1].vertices
-- [3, 4, 1]
```

### See also

`face`, `vertexList` (mesh deform), `generateNormals()`

## video (QuickTime, AVI)

### Syntax

```
member(whichCastMember).video
the video of member whichCastMember
```

### Description

Digital video cast member property; determines whether the graphic image of the specified digital video cast member plays (TRUE or 1) or not (FALSE or 0).

Only the visual element of the digital video cast member is affected. For example, when `video` is set to FALSE, the digital video's soundtrack, if present, continues to play.

**Example**

This statement turns off the video associated with the cast member Interview:

```
member("Interview").video = FALSE
```

**See also**

setTrackEnabled, trackEnabled

## video (RealMedia)

**Syntax**

```
sprite(whichSprite).video  
member(whichCastmember).video
```

**Description**

RealMedia property; allows you to set or get whether the sprite or cast member renders video (TRUE or 1) or only plays the sounds (FALSE or 0). Integer values other than 1 or 0 are treated as TRUE.

Use this property to suppress the video while playing the audio component of a RealMedia cast member, or to toggle the video on and off during playback.

**Examples**

The following examples show that the video property for sprite 2 and the cast member Real is set to TRUE.

```
put sprite(2).video  
-- 1  
  
put member("Real").video  
-- 1
```

The following examples set the video property to FALSE for the RealMedia video element of sprite 2 and the cast member Real.

```
sprite(2).video = FALSE  
member("Real").video = FALSE
```

## videoForWindowsPresent

**Syntax**

```
the videoForWindowsPresent
```

**Description**

System property; indicates whether AVI software is present on the computer.

This property can be tested but not set.

**Example**

This statement checks whether Video for Windows is missing and branches the playhead to the Alternate Scene marker if it isn't:

```
if the videoForWindowsPresent= FALSE then go to "Alternate Scene"
```

**See also**

quickTimeVersion()

## viewH

### Syntax

```
sprite(whichVectorOrFlashSprite).viewH  
the viewH of sprite whichVectorOrFlashSprite  
member(whichVectorOrFlashMember).viewH  
the viewH of member whichVectorOrFlashMember
```

### Description

Cast member and sprite property; controls the horizontal coordinate of a Flash movie and vector shape's view point, specified in pixel units. The values can be floating-point numbers. The default value is 0.

A Flash movie's view point is set relative to its origin point.

Setting a positive value for `viewH` shifts the movie to the left inside the sprite; setting a negative value shifts the movie to the right. Therefore, changing the `viewH` property can have the effect of cropping the movie or even of removing the movie from view entirely.

This property can be tested and set.

**Note:** This property must be set to the default value if the `scaleMode` property is set to `#autoSize`, or the sprite will not display correctly.

### Example

This handler accepts a sprite reference as a parameter and moves the view of a Flash movie sprite from left to right within the sprite's bounding rectangle:

```
on panRight whichSprite  
  repeat with i = 120 down to -120  
    sprite(whichSprite).viewH = i  
    updateStage  
  end repeat  
end
```

### See also

`scaleMode`, `viewV`, `viewPoint`, `viewScale`

## viewPoint

### Syntax

```
sprite(whichVectorOrFlashSprite).viewPoint  
the viewPoint of sprite whichVectorOrFlashSprite  
member(whichVectorOrFlashMember).viewPoint  
the viewPoint of member whichVectorOrFlashMember
```

### Description

Cast member property and sprite property; controls the point within a Flash movie or vector shape that is displayed at the center of the sprite's bounding rectangle in pixel units. The values are integers.

Changing the view point of a cast member changes only the view of a movie in the sprite's bounding rectangle, not the location of the sprite on the Stage. The view point is the coordinate within a cast member that is displayed at the center of the sprite's bounding rectangle and is always expressed relative to the movie's origin (as set by the `originPoint`, `originH`, and `originV` properties). For example, if you set a Flash movie's view point at point (100,100), the center of the sprite is the point within the Flash movie that is 100 Flash movie pixel units to the right and 100 Flash movie pixel units down from the origin point, regardless of where you move the origin point.

The `viewPoint` property is specified as a Director point value: for example, point (100,200). Setting a Flash movie's view point with the `viewPoint` property is the same as setting the `viewH` and `viewV` properties separately. For example, setting the `viewPoint` property to point (50,75) is the same as setting the `viewH` property to 50 and the `viewV` property to 75.

Director point values specified for the `viewPoint` property are restricted to integers, whereas `viewH` and `viewV` can be specified with floating-point numbers. When you test the `viewPoint` property, the point values are truncated to integers. As a general guideline, use the `viewH` and `viewV` properties for precision; use the `originPoint` property for speed and convenience.

This property can be tested and set. The default value is point (0,0).

**Note:** This property must be set to the default value if the `scaleMode` property is set to `#autoSize`, or the sprite will not display correctly.

#### Example

This handler makes a specified Flash movie sprite move down and to the right in increments of five Flash movie pixel units:

```
on panAcross whichSprite
  repeat with i = 1 to 10
    sprite(whichSprite).viewPoint = sprite(whichSprite).viewPoint + point(i *
    -5, i * -5)
    updateStage
  end repeat
end
```

#### See also

`scaleMode`, `viewV`, `viewH`, `viewScale`

## viewScale

### Syntax

`sprite(whichVectorOrFlashSprite).viewScale`  
the `viewScale` of sprite *whichVectorOrFlashSprite*  
`member(whichVectorOrFlashMember).viewScale`  
the `viewScale` of member *whichVectorOrFlashMember*

### Description

Cast member property and sprite property; sets the overall amount to scale the view of a Flash movie or vector shape sprite within the sprite's bounding rectangle. You specify the amount as a percentage using a floating-point number. The default value is 100.

The sprite rectangle itself is not scaled; only the view of the cast member within the rectangle is scaled. Setting the `viewScale` property of a sprite is like choosing a lens for a camera. As the `viewScale` value decreases, the apparent size of the movie within the sprite increases, and vice versa. For example, setting `viewScale` to 200% means the view inside the sprite will show twice the area it once did, and the cast member inside the sprite will appear at half its original size.



One significant difference between the `viewScale` and `scale` properties is that `viewScale` always scales from the center of the sprite's bounding rectangle, whereas `scale` scales from a point determined by the Flash movie's `originMode` property.

This property can be tested and set.

**Note:** This property must be set to the default value if the `scaleMode` property is set to `#autoSize`, or the sprite will not display correctly.

### Example

This sprite script sets up a Flash movie sprite and doubles its view scale:

```
on beginSprite me
    sprite(spriteNum of me).viewScale = 200
end
```

### See also

`scaleMode`, `viewV`, `viewPoint`, `viewH`

## viewV

### Syntax

```
sprite(whichVectorOrFlashSprite).viewV  
the viewV of sprite whichVectorOrFlashSprite  
member(whichVectorOrFlashMember).viewV  
the viewV of member whichVectorOrFlashMember
```

### Description

Cast member and sprite property; controls the vertical coordinate of a Flash movie and vector shape's view point, specified in pixel units. The values can be floating-point numbers. The default value is 0.

A Flash movie's view point is set relative to its origin point.

Setting a positive value for `viewV` shifts the movie up inside the sprite; setting a negative value shifts the movie down. Therefore, changing the `viewV` property can have the effect of cropping the movie or even of removing the movie from view entirely.

This property can be tested and set.

**Note:** This property must be set to the default value if the `scaleMode` property is set to `#autoSize`, or the sprite will not display correctly.

### Example

This handler accepts a sprite reference as a parameter and moves the view of a Flash movie sprite from the top to the bottom within the sprite's bounding rectangle:

```
on panDown whichSprite
    repeat with i = 120 down to -120
        sprite(whichSprite).viewV = i
        updateStage
    end repeat
end
```

### See also

`scaleMode`, `viewV`, `viewPoint`, `viewH`

## visible (sprite property)

### Syntax

`sprite(whichSprite).visible`  
the visible of sprite *whichSprite*

### Description

Sprite property; determines whether the sprite specified by *whichSprite* is visible (TRUE) or not (FALSE). This property affects all sprites in the channel, regardless of their position in the Score.

**Note:** Setting the `visible` property of a sprite channel to FALSE makes the sprite invisible and prevents only the mouse-related events from being sent to that channel. The `beginSprite`, `endSprite`, `prepareFrame`, `enterFrame`, and `exitFrame` events continue to be sent regardless of the sprite's visibility setting. Clicking the Mute button on that channel in the Score, however, will set the `visible` property to FALSE and prevent all events from being sent to that channel. Muting disables a channel, while setting a sprite's `visible` property to FALSE merely affects a graphic property.

This property can be tested and set. If set to FALSE, this property will not automatically reset to TRUE when the sprite ends. You must set the `visible` property of the sprite to TRUE in order to see any other members using that channel.

### Example

This statement makes sprite 8 visible:

```
sprite(8).visible = TRUE
```

## visible (window property)

### Syntax

`window whichWindow.visible`  
the visible of window *whichWindow*

### Description

Window property; determines whether the window specified by *whichWindow* is visible (TRUE) or not (FALSE).

This property can be tested and set.

### Example

This statement makes the window named `Control_Panel` visible:

```
window("Control_Panel").visible = TRUE
```

## visibility

### Syntax

`member(whichCastmember).model(whichModel).visibility`  
`modelObjectReference.visibility`

### Description

3D property; allows you to get or set the `visibility` property of the referenced model. This property determines how the model's geometry is drawn. It can have one of the following values:

- `#none` specifies that no polygons are drawn and the model is invisible.
- `#front` specifies that only those polygons facing the camera are drawn. This method is referred to as back face culling and optimizes rendering speed. This is the default setting for the property.

- `#back` specifies that only those polygons facing away from the camera are drawn. Use this setting when you want to draw the inside of a model, or for models which are not drawing correctly, possibly because they were imported from a file format that used a different handedness value when computing normals.
- `#both` specifies that both sides of all polygons are drawn. Use this setting when you want to see the plane regardless of the viewing direction, and for models that are not drawing correctly.

#### Example

The following statement shows that the visibility property of the model `Monster02` is set to `#none`. The model is invisible.

```
put_member("3D").model("Monster02").visibility
-- #none
```

## voiceCount()

#### Syntax

```
voiceCount()
```

#### Description

Function: returns the number of installed voices available to the text-to-speech engine. The return value is an integer. This number of voices can be used with `voiceSet()` and `voiceGet()` to specify a particular voice to be active.

#### Example

This statement sets the variable `numVoices` to the number of available text-to-speech voices:

```
numVoices = voiceCount()
```

#### See also

`voiceInitialize()`, `voiceSet()`, `voiceGet()`

## voiceGet()

#### Syntax

```
voiceGet()
```

#### Description

Function; returns a property list describing the current voice being used for text-to-speech. The list contains the following properties:

- `#name` indicates the name of the installed voice.
- `#age` indicates the age of the voice. The value is a string. Possible values include “Teen”, “Adult”, “Toddler”, and “Senior”, as well as numeric values such as “35”. Actual values depend on the operating system, speech software version, and voices installed.
- `#gender` indicates whether the voice is male or female. The value is a string.
- `#index` indicates the position of the voice in the list of installed voices. You can refer to a voice by its index when using the `voiceSet()` command.

Use `voiceCount()` to determine the number of available voices.

### Examples

This statement sets the variable `oldVoice` to the property list describing the current text-to-speech voice:

```
oldVoice = voiceGet()
```

This statement displays the property list of the current text-to-speech voice:

```
put voiceGet()  
-- [#name: "Mary", #age: "teen", #gender: "female", #index: 5]
```

### See also

`voiceInitialize()`, `voiceCount()`, `voiceSet()`, `voiceGet()`

## voiceGetAll()

### Syntax

```
voiceGetAll()
```

### Description

Function; returns a list of the available voices installed on the computer. The list is composed of property lists, one for each available voice.

Each property list contains the following properties:

- `#name` indicates the name of the installed voice.
- `#age` indicates the age of the voice. The value is a string. Possible values include “Teen”, “Adult”, “Toddler”, and “Senior”, as well as numeric values such as “35”. Actual values depend on the operating system, speech software version, and voices installed.
- `#gender` indicates whether the voice is male or female.
- `#index` indicates the position of the voice in the list of installed voices. You can refer to a voice by its index when using the `voiceSet()` command.

You can also use `voiceCount()` to determine the number of available voices.

### Examples

This statement sets the variable `currentVoices` to the list of voices installed on the user’s computer:

```
currentVoices = voiceGetAll()
```

This statement displays the property list describing each of the currently installed text-to-speech voices:

```
put voiceGetAll()  
-- [[#name: "Mary", #age: "teen", #gender: "female", #index: 1], [#name: "Joe",  
  #age: "adult", #gender: "male", #index: 2]]
```

### See also

`voiceInitialize()`, `voiceCount()`, `voiceSet()`, `voiceGet()`

## voiceGetPitch()

### Syntax

```
voiceGetPitch()
```

### Description

Function; returns the current pitch for the current voice as an integer. The valid range of values depends on the operating system platform and text-to-speech software.

### Example

These statements check whether the pitch of the current voice is above 10 and set it to 10 if it is:

```
if voiceGetPitch() > 10 then
    voiceSetPitch(10)
end if
```

### See also

`voiceSpeak()`, `voicePause()`, `voiceResume()`, `voiceStop()`, `voiceGetRate()`, `voiceSetRate()`, `voiceSetPitch()`, `voiceGetVolume()`, `voiceSetVolume()`, `voiceState()`, `voiceWordPos()`

## voiceGetRate()

### Syntax

```
voiceGetRate()
```

### Description

Function; returns the current playback rate of the text-to-speech engine. The return value is an integer. The valid range of values depends on the text-to-speech software and operating system platform. In general, values between -10 and 10 can be expected.

### Example

These statements check whether the rate of speech synthesis is below 50 and set it to 50 if it is:

```
if voiceGetRate() < 50 then
    voiceSetRate(50)
end if
```

### See also

`voiceSpeak()`, `voicePause()`, `voiceResume()`, `voiceStop()`, `voiceSetRate()`, `voiceGetPitch()`, `voiceSetPitch()`, `voiceGetVolume()`, `voiceSetVolume()`, `voiceState()`, `voiceWordPos()`

## voiceGetVolume()

### Syntax

```
voiceGetVolume()
```

### Description

Function; returns the current volume of the text-to-speech synthesis. The value returned is an integer. The valid range of values depends on the operating system platform.

**Example**

These statements check whether the text-to-speech volume is at least 55 and set it to 55 if is lower:

```
if voiceGetVolume() < 55 then
    voiceSetVolume(55)
end if
```

**See also**

voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceGetRate(), voiceSetRate(), voiceGetPitch(), voiceSetPitch(), voiceSetVolume(), voiceState(), voiceWordPos()

## voiceInitialize()

**Syntax**

```
voiceInitialize()
```

**Description**

Command; loads the computer's text-to-speech engine. If the voiceInitialize() command returns 0, text-to-speech software is not present or failed to load.

The command returns 1 if successful, 0 otherwise.

**Example**

These statements load the computer's text-to-speech engine and then test for whether the text-to-speech engine has completed loading before using the voiceSpeak() command to speak the phrase "Welcome to Shockwave.":

```
err = voiceInitialize()
if err = 1 then
    voiceSpeak("Welcome to Shockwave")
else
    alert "Text-tospeech software failed to load."
end if
```

**See also**

voiceCount(), voiceSet(), voiceGet()

## voicePause()

**Syntax**

```
voicePause()
```

**Description**

Command; pauses the speech output to the text-to-speech engine. The command returns a value of 1 if it is successful, or 0 if it is not.

**Example**

These statements cause the text-to-speech engine to pause when the user clicks the mouse:

```
on mouseUp
    voicePause()
end mouseUp
```

**See also**

voiceSpeak(), voiceResume(), voiceStop(), voiceGetRate(), voiceSetRate(), voiceGetPitch(), voiceSetPitch(), voiceGetVolume(), voiceSetVolume(), voiceState(), voiceWordPos()

## voiceResume()

### Syntax

```
voiceResume()
```

### Description

Command; resumes the speech output to the text-to-speech engine. The command returns a value of 1 if it is successful, or 0 if it is not.

### Example

These statements resume the speech when the playhead moves to the next frame in the Score:

```
on exitFrame  
    voiceResume()  
end exitFrame
```

### See also

`voiceSpeak()`, `voicePause()`, `voiceStop()`, `voiceGetRate()`, `voiceSetRate()`,  
`voiceGetPitch()`, `voiceSetPitch()`, `voiceGetVolume()`, `voiceSetVolume()`,  
`voiceState()`, `voiceWordPos()`

## voiceSet()

### Syntax

```
voiceSet(integer)
```

### Description

Command; Sets the current voice of the text-to-speech synthesis. The value specified must be an integer. The valid range of values depends on the number of voices installed on the user's computer. If an out-of-range value is specified, the voice is set to the nearest valid value. If successful, the command returns the new value that was set. Use `voiceCount()` to determine the number of available voices.

### Example

This statement sets the current text-to-speech voice to the third voice installed on the user's computer:

```
voiceSet(3)
```

### See also

`voiceInitialize()`, `voiceCount()`, `voiceGet()`

## voiceSetPitch()

### Syntax

```
voiceSetPitch(integer)
```

### Description

Command; sets the pitch for the current voice of the text-to-speech engine to the specified value. The return value is the new pitch value that has been set. The valid range of values depends on the operating system platform and text-to-speech software.

**Example**

This statement sets the pitch for the current voice to 75:

```
voiceSetPitch(75)
```

**See also**

```
voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceGetRate(),  
voiceSetRate(), voiceGetPitch(), voiceGetVolume(), voiceSetVolume(), voiceState(),  
voiceWordPos()
```

## voiceSetRate()

**Syntax**

```
voiceSetRate(integer)
```

**Description**

Command; sets the playback rate of the text-to-speech engine to the specified integer value. The command returns the new value that has been set. The valid range of values depends on the operating system platform. In general, values between -10 and 10 are appropriate for most text-to-speech software. If an out-of-range value is specified, the rate will be set to the nearest valid value.

**Example**

This statement sets the playback rate of the text-to-speech engine to 7:

```
voiceSetRate(7)
```

**See also**

```
voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceGetRate(),  
voiceGetPitch(), voiceSetPitch(), voiceGetVolume(), voiceSetVolume(),  
voiceState(), voiceWordPos()
```

## voiceSetVolume()

**Syntax**

```
voiceSetVolume(integer)
```

**Description**

Command; sets the volume of the text-to-speech synthesis. The specified value must be an integer. The range of valid values depends on the operating system platform. If successful, the command returns the new value that was set. If an invalid value is specified, the volume is set to the nearest valid value.

**Example**

This statement sets the volume of text-to-speech synthesis to 55:

```
voiceSetVolume(55)
```

**See also**

```
voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceGetRate(),  
voiceSetRate(), voiceGetPitch(), voiceSetPitch(), voiceGetVolume(), voiceState(),  
voiceWordPos()
```



## voiceSpeak()

### Syntax

```
voiceSpeak("string")
```

### Description

Command; causes the specified string to be spoken by the text-to-speech engine. When this command is used, any speech currently in progress is interrupted by the new string.

### Example

This statement causes the text-to-speech engine to speak the string “Welcome to Shockwave”:

```
voiceSpeak("Welcome to Shockwave")
```

See also

```
voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceGetRate(),  
voiceSetRate(), voiceGetPitch(), voiceSetPitch(), voiceGetVolume(),  
voiceSetVolume(), voiceState(), voiceWordPos()
```

## voiceState()

### Syntax

```
voiceState()
```

### Description

Function; returns the current status of the voice as a symbol. The possible return values are *#playing*, *#paused*, and *#stopped*.

### Example

These statements check whether the text-to-speech engine is actively speaking and set the voice to 1 if it is not:

```
if voiceState() <> #playing then  
    voiceSet(1)  
end if
```

See also

```
voiceSpeak(), voicePause(), voiceResume(), voiceStop(), voiceGetRate(),  
voiceSetRate(), voiceGetPitch(), voiceSetPitch(), voiceGetVolume(),  
voiceSetVolume(), voiceWordPos(), voiceSpeak()
```

## voiceStop()

### Syntax

```
voiceStop()
```

### Description

Command; stops the speech output to the text-to-speech engine and empties the text-to-speech buffer. The command returns a value of 1 if it is successful, or 0 if it is not.

### Example

These statements stop the speech when the playhead moves to the next frame in the Score:

```
on exitFrame
    voiceStop()
end exitFrame
```

### See also

`voiceSpeak()`, `voicePause()`, `voiceResume()`, `voiceGetRate()`, `voiceSetRate()`,  
`voiceGetPitch()`, `voiceSetPitch()`, `voiceGetVolume()`, `voiceSetVolume()`,  
`voiceState()`, `voiceWordPos()`, `voiceSpeak()`

## voiceWordPos()

### Syntax

```
voiceWordPos()
```

### Description

Function; returns an integer indicating the position of the word that is currently being spoken within the entire string that contains it. For example, if a cast member containing 15 words is being spoken and the fifth word of the cast member is being spoken when the function is used, the return value is 5.

### Example

The following statements cause the sentence “Hello, how are you?” to be spoken and display the current word position in the Message window. Since the `voiceWordPos()` function is called immediately after the `voiceSpeak()` command is used, the return value will be 1.

```
voiceSpeak("Hello, how are you?")
put voiceWordPos()

-- 1
```

### See also

`voiceSpeak()`, `voicePause()`, `voiceResume()`, `voiceStop()`, `voiceGetRate()`,  
`voiceSetRate()`, `voiceGetPitch()`, `voiceSetPitch()`, `voiceGetVolume()`,  
`voiceSetVolume()`, `voiceState()`, `voiceSpeak()`

## VOID

### Syntax

```
VOID
```

### Description

Constant; indicates the value VOID.

### Example

This statement checks whether the value in the variable `currentVariable` is VOID:

```
if currentVariable = VOID then
    put "This variable has no value"
end if
```

### See also

`voidP()`

## voidP()

### Syntax

`voidP(variableName)`

### Description

Function; determines whether the variable specified by *variableName* has any value. If the variable has no value or is `VOID`, this function returns `TRUE`. If the variable has a value other than `VOID`, this function returns `FALSE`.

### Example

This statement checks whether the variable `answer` has an initial value:

```
put voidP(answer)
```

### See also

`ilk()`, `VOID`

## volume (cast member property)

### Syntax

`member(whichCastMember).volume`  
the volume of member *whichCastMember*

### Description

Shockwave Audio (SWA) cast member property; determines the volume of the specified SWA streaming cast member. Values range from 0 to 255.

This property can be tested and set.

### Example

This statement sets the volume of an SWA streaming cast member to half the possible volume:

```
member("SWAfile").volume = 128
```

## volume (sound channel)

### Syntax

`sound(whichChannel).volume`  
the volume of sound *channelNum*

### Description

System property; determines the volume of the sound channel specified by *channelNum*. Sound channels are numbered 1, 2, 3, and so on. Channels 1 and 2 are the channels that appear in the Score.

The value of the `volume` sound property ranges from 0 (mute) to 255 (maximum volume). A value of 255 indicates the full volume set for the machine, as controlled by the `soundLevel` property, and lower values are scaled to that total volume. This property allows several channels to have independent settings within the available range.

The lower the value of the `volume` sound property, the more static or noise you're likely to hear. Using `soundLevel` may produce less noise, although this property offers less control.

This property does not support dot syntax. Use the syntax exactly as shown here.

To see an example of `volume` (`sound channel`) used in a completed movie, see the Sound Control movie in the Learning/Lingo Examples folder inside the Director application folder.

#### Example

This statement sets the volume of sound channel 2 to 130, which is a medium sound level setting:

```
sound(2).volume = 130
```

#### See also

`fadeIn()`, `fadeOut()`, `soundEnabled`, `soundLevel`, `fadeTo()`, `pan` (sound property)

## volume (sprite property)

#### Syntax

```
sprite(whichSprite).volume  
the volume of sprite whichSprite
```

#### Description

Sprite property; controls the volume of a digital video movie cast member specified by name or number. The values range from 0 to 256. Values of 0 or less mute the sound. Values exceeding 256 are loud and introduce considerable distortion.

#### Example

This statement sets the volume of the QuickTime movie playing in sprite channel 7 to 256, which is the maximum sound volume:

```
sprite(7).volume = 256
```

#### See also

`soundLevel`

## warpMode

#### Syntax

```
sprite(whichQTVRSprite).warpMode  
warpMode of sprite whichQTVRSprite
```

#### Description

QuickTime VR sprite property; specifies the type of warping performed on a panorama.

Possible values are `#full`, `#partial`, and `#none`.

This property can be tested and set. When tested, if the values for the static and motion modes differ, the property's value is the value set for the current mode. When set, the property determines the warping for both the static and motion modes.

#### Example

This sets the `warpMode` of sprite 1 to `#full`:

```
sprite(1).warpMode = #full
```

## width

### Syntax

`member(whichCastMember).width`  
the width of member *whichCastMember*  
`imageObject.width`  
`sprite(whichSprite).width`  
the width of sprite *whichSprite*

### Description

Cast member, image object and sprite property; for vector shape, Flash, animated GIF, bitmap, and shape cast members, determines the width, in pixels, of the cast member specified by *whichCastMember*, *imageObject* or *whichSprite*. Field and button cast members are not affected.

For cast members and image objects, this property can be tested; for sprites, this property can be both tested and set.

Setting this property on bitmap sprites automatically sets the sprite's stretch of sprite property to TRUE.

### Examples

This statement assigns the width of member 50 to the variable height:

```
height = member(50).width
```

This statement sets the width of sprite 10 to 26 pixels:

```
sprite(10).width = 26
```

This statement assigns the width of sprite number i + 1 to the variable howWide:

```
howWide = sprite(i + 1).width
```

### See also

height

## width (3D)

### Syntax

`member(whichCastmember).modelResource(whichModelResource).width`  
`modelResourceObjectReference.width`

### Description

3D property; allows you to get or set the width of the plane for a model resource whose type is #box or #plane. This property must be greater than 0.0, and has a default setting of 1.0. For objects whose type is #box, the default value of width is 50.0. For objects whose type is #plane, the default setting is 1.0. width is measured along the X axis.

### Example

This statement sets the width of the model resource Grass plane to 250.0:

```
member("3D World").modelResource("Grass plane").width = 250.0
```

## widthVertices

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).  
widthVertices  
modelResourceObjectReference.widthVertices
```

### Description

3D property; allows you to get or set the number of vertices (as an integer) on the X axis of a model resource whose type is `#box` or `#plane`. This property must be greater than or equal to 2, and has a default value of 2.

### Example

The following statement sets the `widthVertices` property of the model resource `Tower` to 10. Eighteen polygons ( $2 * (10-1)$  triangles) will be used to define the geometry of the model resource along its *x*-axis.

```
member("3D World").modelResource("Tower").widthVertices = 10
```

## wind

### Syntax

```
member(whichCastmember).modelResource(whichModelResource).wind  
modelResourceObjectReference.wind
```

### Description

3D property; allows you to get or set the `wind` property of a model resource whose type is `#particle`, as a vector.

This `wind` property defines the direction and strength of the wind force applied to all particles during each simulation step. The default value for this property is `vector(0, 0, 0)`, which specifies that no wind is applied.

### Example

```
put member("3D").modelResource("fog bank").wind  
-- vector(10.5,0,0)
```

## window

### Syntax

```
window whichWindow
```

### Description

Keyword; refers to the movie window—a window that contains a Director movie—specified by *whichWindow*.

Windows that play movies are useful for creating floating palettes, separate control panels, and windows of different shapes. Using windows that play movies, you can have several movies open at once and allow them to interact.

### Examples

This statement opens a window named `Navigation`:

```
open window "Navigation"
```

This statement moves the Navigation window to the front:

```
moveToFront window "Navigation"
```

or

```
window("Navigation").moveToFront()
```

**See also**

close window, moveToBack, moveToFront, open window

## windowList

**Syntax**

the windowList

**Description**

System property; displays a list of references to all known movie windows.

**Examples**

This statement displays a list of all known movie windows in the Message window:

```
put the windowList
```

This statement clears windowList:

```
the windowList = [ ]
```

**See also**

windowPresent()

## windowPresent()

**Syntax**

```
windowPresent(windowName)
```

**Description**

Function; indicates whether the object specified by *windowName* is running as a movie in a window (TRUE) or not (FALSE). If a window had been opened, windowPresent remains TRUE for the window until the window has been removed from the windowList property.

The *windowName* argument must be the window's name as it appears in the windowList property.

**Example**

This statement tests whether the object myWindow is a movie in a window (MIAW) and then displays the result in the Message window:

```
put windowPresent(myWindow)
```

**See also**

windowList

# windowType

## Syntax

*window* *whichWindow*.windowType  
the windowType of window *whichWindow*

## Description

Window property; controls the display style of the window specified by *whichWindow*, as follows:

- 0—Movable, sizable window without zoom box
- 1—Alert box or modal dialog box
- 2—Plain box, no title bar
- 3—Plain box with shadow, no title bar
- 4—Movable window without size box or zoom box
- 5—Movable modal dialog box
- 8—Standard document window
- 12—Zoomable, nonresizable window
- 16—Rounded-corner window
- 49—Floating palette, during authoring (in Macintosh projectors, the value 49 specifies a stationary window)

You can set this property before opening the window. Numbers 6, 7, 9, 10, 11, 13, 14, 15, and 17 through 48 have no effect when used as the windowType value.

You can change the windowType setting after a window has been opened, but a delay may occur while the window is redrawn.

If no windowType setting is specified, a value of 0 is used.

In Microsoft Windows, these numbers create windows with the same functionality just described, but with a Windows appearance. Other values for windowType are possible, but use them with caution, because some modal windows can be exited only when you restart the computer.

## Example

This statement sets the value of the display style of the window named “Control\_Panel” to 8:

```
window("Control_Panel").windowType = 8
```

# word...of

## Syntax

*member*(*whichCastMember*).word[*whichWord*]  
*textMemberExpression*.word[*whichWord*]  
*chunkExpression*.word[*whichWord*]  
word *whichWord* of *fieldOrStringVariable*  
*fieldOrStringVariable*. word[*whichWord*]  
*textMemberExpression*.word[*firstWord*..*lastWord*]  
*member*(*whichCastMember*).word[*firstWord*..*lastWord*]  
word *firstWord* to *lastWord* of *chunkExpression*  
*chunkExpression*.word[*whichWord*..*lastWord*]



### Description

Chunk expression; specifies a word or a range of words in a chunk expression. A word chunk is any sequence of characters delimited by spaces. (Any nonvisible character, such as a tab or carriage return, is considered a space.)

The expressions *whichWord*, *firstWord*, and *lastWord* must evaluate to integers that specify a word in the chunk.

Chunk expressions refer to any character, word, item, or line in any source of characters. Sources of characters include field and text cast members and variables that hold strings.

To see an example of `word...of` used in a completed movie, see the Text movie in the Learning/Lingo Examples folder inside the Director application folder.

### Examples

These statements set the variable named `animalList` to the string “fox dog cat” and then insert the word *elk* before the second word of the list:

```
animalList = "fox dog cat"  
put "elk" before animalList.word[2]
```

The result is the string “fox elk dog cat”.

This statement tells Director to display the fifth word of the same string in the Message window:

```
put "fox elk dog cat".word[5]
```

Because there is no fifth word in this string, the Message window displays two quotation marks (“”), which indicate an empty string.

### See also

`char...of`, `line...of`, `item...of`, `count()`, `number (words)`

## wordWrap

### Syntax

`member(whichCastMember).wordWrap`  
the `wordWrap` of member *whichCastMember*

### Description

Cast member property; determines whether line wrapping is allowed (TRUE) or not (FALSE).

### Example

This statement turns line wrapping off for the field cast member Rokujo:

```
member("Rokujo").wordWrap = FALSE
```

## worldPosition

### Syntax

```
member(whichCastmember).model(whichModel).worldPosition  
member(whichCastmember).light(whichLight).worldPosition  
member(whichCastmember).camera(whichCamera).worldPosition  
member(whichCastmember).group(whichGroup).worldPosition
```

### Description

3D property; allows you to get and not set the position of the specified node in world coordinates. A node can be a model, group, camera, or light. This property is equivalent in result to using `getWorldTransform().position` command. The position of a node is represented by a vector object.

### Example

This statement shows that the position of the model named Mars, in world coordinates, is the vector (-1333.2097, 0.0000, -211.0973):

```
put member("scene").model("Mars").worldPosition  
-- vector(-1333.2097, 0.0000, -211.0973)
```

### See also

`getWorldTransform(), position (transform)`

## worldSpaceToSpriteSpace

### Syntax

```
member(whichCastmember).camera(whichCamera).worldSpaceToSprite\  
Space(vector)
```

### Description

3D command; returns the point within the camera's rect at which the world-relative position specified by *vector* would appear. The position returned by this command is relative to the upper left corner of the camera's rect.

If the position specified is out of view of the camera, this command returns `void`.

### Example

This statement shows that the world origin, specified by vector (0, 0, 0), appears at point (250,281) within the camera's rect:

```
put sprite(5).camera.worldSpaceToSpriteSpace(vector(0, 0, 0))  
-- point(250, 281)
```

### See also

`spriteSpaceToWorldSpace, rect (camera)`

## worldTransform

### Syntax

```
member(whichMember).model(whichModel).bonesPlayer.bone[index].\  
worldTransform
```

### Description

3D bonesplayer property; allows you to get the world relative transform of a specific bone, as opposed to using the `transform` property which returns the bone's parent relative transform. The `worldTransform` property can only be used with bonesplayer modified models.

### Example

This statement stores a bone's world relative transform in the variable `finalTransform`:

```
finalTransform =  
member("3D").model("biped").bonesPlayer.bone[3].worldTransform
```

### See also

`bone`, `getWorldTransform()`, `transform` (property)

## wrapTransform

### Syntax

```
member( whichCastmember ).shader( ShaderName ).wrapTransform  
member( whichCastmember ).shader[ ShaderIndex ].wrapTransform  
member( whichCastmember ).model[modelName].shader.wrapTransform  
member( whichCastmember ).model.shaderlist[ shaderListIndex ].\  
wrapTransform
```

### Description

3D standard shader property; this property provides access to a transform that modifies the texture coordinate mapping for the shader's texture. Rotate this transform to alter how the texture is projected onto a model surface. The texture remains unaffected; the transform modifies only the orientation of how the shader applies the texture.

**Note:** Note: This command only has an effect when the shader's `textureModelList` is set to is `#planar`, `#spherical`, or `#cylindrical`.

### Example

These statements set the `transformMode` of the shader named "shad2" to `#wrapCylindrical`, then rotates that cylindrical projection by 90° about the *x*-axis so that the cylindrical mapping wraps around the *y*-axis instead of the *z*-axis:

```
s = member("Scene").shader("shad2")  
s.textureMode= #wrapCylindrical  
s.wrapTransform.rotate(90.0, 0.0, 0.0)
```

## wrapTransformList

### Syntax

```
member( whichCastmember ).shader( ShaderName ).wrapTransformList\  
[ textureLayerIndex ]  
member( whichCastmember ).shader[ shaderListIndex ].\  
wrapTransformList[ textureLayerIndex ]  
member( whichCastmember ).model( modelName ).\  
shader.wrapTransformList[ textureLayerIndex ]  
member( whichCastmember ).model( modelName ).shaderList\  
[ shaderListIndex ]. wrapTransformList[ textureLayerIndex ]
```

### Description

3D standard shader property; this property provides access to a transform that modifies the texture coordinate mapping of a specified texture layer. Rotate this transform to alter how the texture is projected onto model surfaces. The texture itself remains unaffected; the transform modifies only the orientation of how the shader applies the texture.

**Note:** wrapTransformList[textureLayerIndex] only has an effect when textureModelList[textureLayerIndex] is set to #planar, #spherical, or #cylindrical.

### Example

In this example, line 2 sets the transformMode of the third texture layer of the shader named “shad2” to #wrapCylindrical. Line 3 rotates that cylindrical projection by 90° about the *x*-axis so that the cylindrical mapping wraps around the *y*-axis instead of the *z*-axis.

```
s = member("Scene").shader("shad2")  
s.textureModelList[3] = #wrapCylindrical  
s.wrapTransformList[3].rotate(90.0, 0.0, 0.0)
```

### See also

newShader, textureModelList

## x (vector property)

### Syntax

```
member(whichCastmember).vector.x  
member(whichCastmember).vector[1]
```

### Description

3D property; allows you to get or set the *x* component of a vector.

### Example

This statement shows the *x* component of a vector:

```
vec = vector(20, 30, 40)  
put vec.x  
-- 20.0000
```

## xAxis

### Syntax

`member(whichCastmember).transform.xAxis`

### Description

3D transform property; allows you to get but not set the vector representing the transform's canonical *x*-axis in transform space.

### Example

The first line of this example sets the transform of the model ModCylinder to the identity transform. The next two lines show that the *x*-axis of ModCylinder is the vector ( 1.0000, 0.0000, 0.0000 ). This means that the *x*-axis of ModCylinder is aligned with the *x*-axis of the world. The next line rotates ModCylinder 90° around its *y*-axis. This rotates the axes of ModCylinder as well. The last two lines show that the *x*-axis of ModCylinder is now the vector ( 0.0000, 0.0000, -1.0000 ). This means that the *x*-axis of ModCylinder now is aligned with the negative *z*-axis of the world.

```
member("Engine").model("ModCylinder").transform.identity()  
put member("Engine").model("ModCylinder").transform.xAxis  
-- vector( 1.0000, 0.0000, 0.0000 )  
member("Engine").model("ModCylinder").rotate(0, 90, 0)  
put member("Engine").model("ModCylinder").transform.xAxis  
-- vector( 0.0000, 0.0000, -1.0000 )
```

## xtra

### Syntax

`xtra whichXtra`

### Description

Function; returns an instance of the scripting Xtra specified by *whichXtra*.

To see an example of `xtra` used in a completed movie, see the Read and Write Text movie in the Learning/Lingo Examples folder inside the Director application folder.

### Example

This statement uses the `new()` function to create a new instance of the Multiuser Xtra and assigns it to the variable `tool`:

```
tool = new(xtra "Multiuser")
```

### See also

`new()`

# xtraList

## Syntax

the xtraList

## Description

System property; displays a linear property list of all available Xtra extensions and their file versions. This property is useful when the functionality of a movie depends on a certain version of an Xtra.

There are two possible properties that can appear in XtraList:

---

#name	Specifies the filename of the Xtra on the current platform. It is possible to have a list without a #name entry, such as when the Xtra exists only on one platform.
#version	Specifies the same version number that appears in the Properties dialog box (Windows) or Info window (Macintosh) when the file is selected on the desktop. An Xtra may not necessarily have a version number.

---

## Examples

This statement displays the xtraList in the Message window:

```
put the xtraList
```

This handler checks the version of a given Xtra:

```
-- This handler checks all the available Xtra extensions to return the version
-- of the requested Xtra
-- The XtraFileName may be only a partial match if desired
-- Use the full filename for more exact usage
-- The string returned is either the version property for the Xtra or an empty
-- string
-- It may be empty if either the version property doesn't exist or the Xtra is
-- not found in the available list
on GetXtraVersion XtraFileName
    -- Get the entire list of Xtra extensions and their information
    listOfXtras = the xtraList
    -- Initialize the local variable to contain the version
    theVersion = ""
    -- Iterate through all the Xtra extensions listed
    repeat with currentXtra in listOfXtras
        -- If the current Xtra's name contains the Xtra passed in, then check for
        -- the version
        if currentXtra.name contains XtraFileName then
            -- First determine if the version property exists for that Xtra
            versionFlag = getaProp(currentXtra, #version)
            -- If the version property is not VOID, then set the local variable
            to
            -- the string contained in that property value
            if not voidP(versionFlag) then
                theVersion = currentXtra.version
            end if
        end if
    end repeat
    -- Return the version information found or an empty string
    return theVersion
end
```

## See also

movieXtraList, getaProp

## xtras

### See

number of xtras

## y (vector property)

### Syntax

```
member(whichCastmember).vector.y  
member(whichCastmember).vector[2]
```

### Description

3D property; allows you to get or set the *y* component of a vector.

### Example

This statement shows the *y* component of a vector:

```
vec = vector(20, 30, 40)  
put vec.y  
-- 30.0000
```

## yAxis

### Syntax

```
member(whichCastmember).transform.yAxis
```

### Description

3D transform property; allows you to get but not set the vector representing the transform's canonical *y*-axis in transform space.

### Example

The first line of this example sets the transform of the model `ModCylinder` to the identity transform. The next two lines show that the *Y* axis of `ModCylinder` is the vector ( 0.0000, 1.0000, 0.0000 ). This means that the *y*-axis of `ModCylinder` is aligned with the *y*-axis of the world. The next line rotates `ModCylinder` 90° around its *x*-axis. This rotates the axes of `ModCylinder` as well. The last two lines show that the *y*-axis of `ModCylinder` is now the vector ( 0.0000, 0.0000, 1.0000 ). This means that the *y*-axis of `ModCylinder` now is aligned with the positive *z*-axis of the world.

```
member("Engine").model("ModCylinder").transform.identity()  
put member("Engine").model("ModCylinder").transform.yAxis  
-- vector( 0.0000, 1.0000, 0.0000 )  
member("Engine").model("ModCylinder").rotate(90, 0, 0)  
put member("Engine").model("ModCylinder").transform.yAxis  
-- vector( 0.0000, 0.0000, 1.0000 )
```

## yon

### Syntax

```
member(whichCastmember).camera(whichCamera).yon
```

### Description

3D property; allows you to get or set the distance from the camera defining where along the camera's Z axis the view frustum is clipped. Objects at a distance greater than `yon` are not drawn.

The default value for this property is 3.40282346638529e38.

### Example

This statement sets the `yon` property of camera 1 to 50000:

```
member("3d world").camera[1].yon = 50000
```

### See also

`hither`

## z (vector property)

### Syntax

```
member(whichCastmember).vector.z  
member(whichCastmember).vector[3]
```

### Description

3D property; allows you to get or set the `z` component of a vector.

### Example

This statement shows the `z` component of a vector:

```
vec = vector(20, 30, 40)  
put vec.z  
-- 40.0000
```

## zAxis

### Syntax

```
member(whichCastmember).transform.zAxis
```

### Description

3D transform property; allows you to get but not set the vector representing the transform's canonical `z`-axis in transform space.



### Example

The first line of this example sets the transform of the model `ModCylinder` to the identity transform. The next two lines show that the  $z$ -axis of `ModCylinder` is the vector ( 0.0000, 0.0000, 1.0000). This means that the  $z$ -axis of `ModCylinder` is aligned with the  $z$ -axis of the world. The next line rotates `ModCylinder` 90° around its  $y$ -axis. This rotates the axes of `ModCylinder` as well. The last two lines show that the  $z$ -axis of `ModCylinder` is now the vector ( 1.0000, 0.0000, 0.0000 ). This means that the  $z$ -axis of `ModCylinder` now is aligned with the  $x$ -axis of the world.

```
member("Engine").model("ModCylinder").transform.identity()
put member("Engine").model("ModCylinder").transform.zAxis
-- vector( 1.0000, 0.0000, 0.0000 )
member("Engine").model("ModCylinder").rotate(0, 90, 0)
put member("Engine").model("ModCylinder").transform.zAxis
-- vector( 0.0000, 0.0000, -1.0000 )
```

## zoomBox

### Syntax

```
zoomBox startSprite, endSprite {, delayTicks}
```

### Description

Command; creates a zooming effect, like the expanding windows in the Macintosh Finder. The zoom effect starts at the bounding rectangle of *startSprite* and finishes at the bounding rectangle of *endSprite*. The `zoomBox` command uses the following logic when executing:

- 1 Look for *endSprite* in the current frame; otherwise,
- 2 Look for *endSprite* in the next frame.

Note, however, that the `zoomBox` command does not work for *endSprite* if it is in the same channel as *startSprite*.

The *delayTicks* variable is the delay in ticks between each movement of the zoom rectangles. If *delayTicks* is not specified, the delay is 1.

### Example

This statement creates a zoom effect between sprites 7 and 3:

```
zoomBox 7, 3
```

## on zoomWindow

### Syntax

```
on zoomWindow
    statement(s)
end
```

### Description

System message and event handler; contains statements that execute whenever a movie running as a movie in a window (MIAW) is resized. This happens when the user clicks the Minimize or Maximize button (Windows) or the Zoom button (Macintosh). The operating system determines the dimensions after resizing the window.

An `on zoomWindow` event handler is a good place to put Lingo that rearranges sprites when window dimensions change.

**Example**

This handler moves sprite 3 to the coordinates stored in the variable `centerPlace` when the window that the movie is playing in is resized:

```
on zoomWindow
    centerPlace = point(10, 10)
    sprite(3).loc = centerPlace
end
```

**See also**

`drawRect`, `sourceRect`, `on_resizeWindow`

# INDEX

## Symbols

- " (quotation mark) 530
- " (string constant) 54
- # (symbol definition operator) 43, 525
- & (concatenation operator) 45
- && (concatenation operator) 46
- () (parentheses operator) 47
- \* (multiplication operator) 47
- + (addition operator) 48
- (subtraction operator) 44
- (comment delimiter) 45
- / (division operator) 49
- < (less than operator) 50
- <= (less than or equal to operator) 50
- <> (not equal operator) 50
- = (equal sign operator) 51
- > (greater than operator) 51
- >= (greater than or equal to operator) 51
- @ (pathname operator) 54
- [] (list brackets) 52
- ¬ (continuation symbol) 54

## Numerics

- 3D capabilities, determining with Lingo 37
- 3D text, new Lingo for 41

## A

- abbreviated elements 56
- abbreviated time function 677
- active sprites 138
- adding
  - to linear lists 60, 65, 77
  - to property lists 65
- addition operator (+) 48
- AIFF files 624
- alert command 66
- alertHook event handler 219
- alignment of member field property 68

- allowZooming movie property 71
- Alt key (Windows) 456
- ampersand operators (& or &&) 45, 46
- ancestor, sending messages to 117
- and logical operator 73
- animation, Lingo for 31
- applications
  - shutDown command and 613
  - starting 454
- ASCII codes 130, 450
- assigning
  - cast members to sprites 378
  - palettes to cast members 463, 464
- asterisk operator (\*) 47

## B

- backdrops 62, 64, 318
  - manipulating with Lingo 32
- background colors, setting with cast members 84
- background events, processing 163
- BACKSPACE character constant 86
- beep elements
  - alert command 66
  - beep command 87
  - beepOn property 87
- behavior scripts, sprites and 583
- behaviors, attaching 326
- bilinear filtering 423
- bit rates 93
- bitmap cast members
  - color depth of 191
  - palettes associated with 464
  - registration point of 547
  - size of 614
- bitmap compression 313, 414, 415
- bitmaps
  - picture of member property 481, 482
  - trimming white space 699

- blend range, end and start 100
- bonesPlayer modifier 32
- borders
  - of field cast members 105
  - of shape cast members 349
- boxes
  - for field cast members 368
  - Lingo for 38
- brackets ([]) 52
- branching
  - case keyword 121
  - end case keyword 216
  - if...then statements 305
  - otherwise keyword 461
  - repeat while keyword 552
- browsers
  - clearing cache in 134
  - displaying strings in 428
  - location of 110
  - preloading files from Internet 506
  - retrieving files and 274
  - sending strings to 228
  - setting cache size of 115
- button properties
  - buttonType 113
  - checkBoxAccess 131
  - hilite of member 294
- buttons
  - buttonStyle property 113
  - buttonType property 113
  - radio button sprites 294
- C**
- cache
  - clearing in browsers 134
  - refreshing web pages and 115
  - setting size in browsers 115
- cameras
  - adding 62
  - Lingo for 32
- canceling network operations 424
- canonical X axis 745
- canonical Y axis 747, 748
- canonical Z axis 748
- cast members
  - assigned to first sound channel 259
  - assigned to second sound channel 259
  - assigning to sprites 378
  - borders 349
  - canceling loading of 120
  - cast member numbers 242, 444
  - cast members *continued*
    - castLibNum of member 122
    - changing cast member names 419
    - changing cast members used for cursors 173
    - copying 161, 208, 373, 481
    - creating 429
    - cursor command 173
    - deleting 222
    - determining if loaded in memory 351
    - displaying information about when loading 687
    - downloading from Internet 373
    - duration of 208
    - editing 211, 392
    - empty cast members 242
    - font used to display 250
    - height of 291
    - importing files into 314
    - in Cast window 592
    - last cast member in casts 448
    - line spacing for 348
    - lines in 347
    - loading 243, 303, 304
    - locToCharPos function 354
    - locVToLinePos function 356
    - media in cast member tracks 692
    - media of member function 373, 481
    - member keyword 375
    - modification date 393
    - modified by property 393
    - modified of member function 392
    - moving 410
    - names 419
    - palettes associated with 463, 464
    - pasting Clipboard contents into 470
    - pausing playback of 476
    - pictureP function 482
    - preloading 303, 503, 505
    - purge priority 521
    - replacing content of 314
    - setting background colors with 84
    - Shockwave Audio 266, 268
    - size of 614
    - streaming, Lingo for 40
    - text boxes for 108
    - text of script assigned to 585
    - unloading from memory 521, 707, 708
    - width of 737

- cast properties
  - fileName of castLib 236
  - preLoadMode of CastLib 505
  - selection of castLib 592
- Cast window, cast members selected in 592
- casts
  - activeCastLib system property 59
  - cast numbers 444
  - castLib keyword 122
  - castLibNum of member 122
  - file names of 236
  - last cast member in 448
  - names of 419
  - number of in movies 444
  - preload mode of 505
  - saving changes to 576
- CGI query 272
- channel numbers, sprite behaviors and 630
- channels
  - cast member assigned to first sound channel 259
  - cast member assigned to second sound channel 259
  - channelCount of member property 127
  - checking sound channels 617
  - closing sound channels 618
  - palette channels as puppets 517
  - selecting in Score window 582
  - sound channels as puppets 518
  - tempo channels as puppets 519
- character constants
  - BACKSPACE 86
  - EMPTY 214
  - ENTER 220
  - QUOTE 530
  - RETURN 559
  - TAB 659
- character strings
  - ASCII codes 130, 450
  - char...of keyword 127
  - chars function 129
  - comparing 156
  - converting expressions to 653
  - counting characters in 448
  - counting items in 445
  - counting lines in 446
  - counting words in chunk expressions 448
  - deleting chunk expressions 184
  - do command 200
  - EMPTY character constant 214
  - ending character in selections 593
  - expressions as 653
  - character strings *continued*
    - field keyword 235
    - highlighting 293
    - in field cast members 664
    - integer function 321
    - item...of keyword 328
    - last function 339
    - length function 343
    - line...of keyword 346
    - numerical values of 717
    - offset function 452
    - put...after command 522
    - put...before command 523
    - put...into command 523
    - selecting 592
    - selecting words in 740
    - starting character in selections 594
    - starts comparison operator 637
- characters, locating in field cast members 354, 356
- check boxes
  - check box sprites 294
  - checkBoxAccess property 131
  - checkBoxType property 131
  - hilite of member property 294
- child nodes, Lingo for 33
- child objects
  - ancestor property 72
  - creating 429
  - me keyword 372
  - sending messages to ancestor of 117
  - tracking 59
- chunk expressions
  - char...of keyword 127
  - counting characters in 448
  - counting items in 445
  - counting lines in 446
  - counting words in 448
  - deleting 184
  - field keyword 235
  - highlighting 293
  - item...of keyword 328
  - last function 339
  - line...of keyword 346
  - put...after command 522
  - put...before command 523
  - put...into command 523
  - selecting words in 740
  - word...of keyword 740

- Clipboard
  - copying cast members to 161
  - pasting contents into cast members 470
- closing
  - sound channels 618
  - windows 141, 254
- collisions
  - detection, and Lingo 33
  - resolving 557
- color depth
  - colorDepth property 148
  - depth of member property 191
  - of bitmap cast members 191
  - of monitors 148
  - setting (Macintosh) 657
- colors
  - Score color assigned to sprites 582
  - setting background colors with cast members 84
  - setting foreground colors of field cast members 252
  - setting Stage color 633
- Command key (Macintosh) 153
- comment delimiter (--) 45
- comparing values 121
- comparison operators
  - contains 156
  - equal sign operator (=) 51
  - greater than operator (>) 51
  - greater than or equal to operator (>=) 51
  - less than operator (<) 50
  - less than or equal to operator (<=) 50
  - not equal operator (<>) 50
  - sprite...within 630
  - starts 637
- compression 154
- computers
  - platform system property 482
  - restarting Macintosh computers 558
  - turning off 613
- concatenation operators (& or &&) 45, 46
- conditions
  - comparing 121
  - end case keyword 216
  - if...then statements 305
  - otherwise keyword 461
  - repeat while keyword 552
- constants
  - BACKSPACE 86
  - EMPTY 214
  - ENTER 220
  - FALSE 234
  - constants *continued*
    - QUOTE 530
    - RETURN 559
    - SPACE 626
    - TAB 659
    - TRUE 700
    - VOID 734
- continuation symbol (~) 54
- Control key (Macintosh) 157
- Control key (Windows) 153, 157
- converting
  - ASCII codes to characters 450
  - characters to ASCII codes 130
  - duration in time to frames 295
  - expressions to floating-point numbers 248
  - frames to duration in time 259
  - time to ticks 676
- coordinates
  - of points 491
  - of rectangles 540, 625
  - of sprites 105, 154, 156, 342, 562, 685
  - of windows 541
- Copy command, enabling 212
- copying
  - cast members 161, 208, 373, 481
  - frames 206
  - lists 207
- counting
  - characters in strings 343
  - items in lists 161
  - parameters sent to handler 466
  - tracks on digital video sprites 688
- CPU, processing background events 163
- creating
  - cast members 429
  - child objects 429
  - custom menus 379
  - delimiters for items 329
  - linear lists 350
  - rectangles 538
  - Xtra extensions 429
- cue points
  - list of names 169
  - list of times 170
  - sprites and 327, 395
- cursors
  - changing cast members used for 173
  - cursor command 173
  - cursor of sprite property 175
  - cursor resources 175

cursors *continued*

- identifying character under 396
- mouse pointer position 400, 408, 409
- mouseChar function 396
- mouseMember function 404
- rollOver tests 566

curve, adding 432

Cut command, enabling 212

cylinders

- Lingo for 38
- open and sealed 686

## D

date object 590

debugging

- with put command 522
- with showGlobals command 611
- with showLocals command 611

Delete key (Macintosh) 86

deleting

- all items in list 184
- cast members 222
- chunk expressions 184
- contents of frame's sprite 135
- frames from movies 186
- items in lists 184, 188
- movies from memory 708
- values from lists 188
- windows 254

delimiters

- comment delimiter (--) 45
- defining for items 329

digital video

- deleting frames from movies 186
- duration of 208
- format of 195
- pausing movies 157
- playing tracks on 606
- preloading movies 507

digital video cast member properties

- center of member 125
- controller of member 158
- digitalVideoType of member 195
- frameRate of member 256
- loop of member 360
- timeScale of member property 683
- trackStartTime (member) 690
- trackStopTime (member) 691
- trackType (member) 692
- video of member 721, 726
- volume of sprite 736

digital video cast members

- counting tracks on 688
- media in tracks 692
- Paused At Start of member property 474
- preloading into memory 503
- start time of tracks 690
- stop time of tracks 691
- time scale for 194
- turning play of on or off 721, 726

digital video sprite properties

- trackNextKeyTime 689
- trackNextSampleTime 689
- trackPreviousKeyTime 690
- trackPreviousSampleTime 690
- trackText 692
- trackType (sprite) 693

digital video sprites

- counting tracks on 688
- media in tracks 693
- playing tracks on 689
- text in tracks 692

dimensions of rectangles 316

Director, exiting 530, 613

disabling

- sound 619
- stream status reporting 662

distance, of lines 349

division operator (/) 49

division, remainders of 384

documents, opening applications with 454

double-byte characters 318

downloading cast members from Internet 373

drawing 204

drop shadows of field cast members 108, 205

## E

e logarithm functions 227, 358

editing

- cast members 211
- field sprites 214

EMBED tags, external parameters 229, 230

empty cast members 242

EMPTY character constant 214

enabling sound 619

enabling stream status reporting 662

ending Score recording session 218

engraver shader, Lingo for 40

ENTER character constant 220

Enter key 220

equal sign operator (=) 51

## errors

- during network operations 425
- on alertHook event handler 219
- Shockwave Audio cast members and 266, 268

evaluating expressions 200, 522, 523, 597

## event handlers

- ancestor property 72
- counting parameters sent to 466
- exit repeat keyword 227
- exiting 56, 225, 290
- marking end of 216
- on activateWindow 59
- on alertHook 219
- on closeWindow 219
- on cuePassed 219
- on deactivateWindow 219
- on endSprite 219
- on enterFrame 225, 478
- on exitFrame 225
- on idle 301
- on keyDown 679
- on keyUp 336
- on keyword 454
- on mouseDown 397, 681
- on mouseEnter 402
- on mouseLeave 402
- on mouseUp 408
- on mouseUpOutside 408
- on mouseWithin 408
- on moveWindow 455
- on openWindow 455
- on prepareFrame 508
- on rightMouseUp 564
- on stopMovie 652
- on zoomWindow 454
- result function 558
- return keyword 560
- tell command 661
- timeOut 454

## event messages

- custom 594, 595
- passing 469
- stopping 454

## events

- identifying sprites in 171
- outside of windows 385

## exiting

- Director 530, 613
- handlers 56, 225, 290
- projectors 226

exponents 501

## expressions

- as integers 321
- as strings 653
- as symbols 657, 658
- converting to floating-point numbers 248
- converting to strings 653
- evaluating 200, 522, 523, 597
- FALSE expressions 234
- floating-point numbers in 248
- logical negation of 442

## F

fading in sound 620

fading out sound 621

FALSE logical constant 234

## field cast member properties

- autoTab of member 83
- border of member 105
- boxDropShadow of member 108
- dropShadow of member 205
- editable of member 211
- lineCount of member 347
- margin of member 368
- pageHeight of member 463
- wordWrap of member property 741

## field cast members

- drop shadows for 205
- field keyword 235
- font size of 251
- font style of 251
- height of lines in 347
- installing menus defined in 320
- lineHeight function 347
- lineHeight of member property 348
- locToCharPos function 354
- locVToLinePos function 356
- position of lines in 349
- scrolling 586, 587
- strings in 664
- text boxes for 108

## field properties

- alignment of member 68
- font of member 250
- fontSize of member 251
- fontStyle of member 251
- lineHeight of member 348
- selEnd 593
- selStart 594



- field sprites
  - editing 214
  - mouseChar function 396
- file size of movies 414
- filenames
  - finding 274, 589
  - of casts 236
  - of linked cast members 237
- files
  - preloading from Internet 203, 506
  - retrieving 274
  - retrieving text from over network 272
  - writing strings to 603
- Finder, quitting from Director to (Macintosh) 530
- finding
  - empty cast members 242
  - filenames 274, 589
  - movies 274
- Flash Asset Xtra, endTellTarget command for 663
- Flash movies
  - mixing sounds 622
  - setting properties of 601
  - setting variables in 606
- flipped integer value 423
- floating-point numbers 248, 249
- fog, Lingo for 34
- folders
  - of current movie 415, 472
  - searchCurrentFolder function 589
- fonts 250, 251
- foreground colors, setting with cast members 252
- forward slash (/) 49
- frame properties
  - frameLabel 255
  - framePalette 256
  - frameRate of member 256
  - frameScript 258
  - frameSound1 259
  - frameSound2 259
  - frameTempo 260
  - frameTransition 260
  - lastFrame 341
  - timeScale 683
  - trackNextKeyTime 689
  - trackPreviousKeyTime 690
- frame scripts
  - cast member number of 258
  - rollOver tests 566
- frames
  - clearFrame command 135
  - converting duration in time to 295
  - converting to duration in time 259
  - copying 206
  - deleting 186
  - deleting frame's sprite contents 135
  - displaying number of current frame 254
  - framesToHMS function 259
  - going to 284
  - HMSToFrames function 295
  - inserting 319
  - labels assigned to 255
  - listing frame labels 338
  - marker function 368
  - marker labels and 338
  - markers before and after 368
  - memory needed to display 531
  - number of palette in 256
  - on enterFrame event handler 225
  - on exitFrame event handler 225
  - on prepareFrame event handler 508
  - playing 483
  - printing 513
  - tempo settings 260
  - transitions between 260, 520
  - updating 709
- free memory 261, 379, 413, 531
- front slash (/) 49
- FTP proxy servers, setting values of 516

**G**

- generating random numbers 533
- global properties, cpuHogTicks 163
- global variables 136, 282, 523, 611
- greater than operator (>) 51
- greater than or equal to operator (>=) 51
- grouping operator () 47
- groups (3D), Lingo for 34

**H**

- handlers
  - counting parameters sent to 466
  - exiting 56, 225, 290
  - invoking 116
  - marking end of 216
- hardware information, getting 270

- height
  - height of member property 291
  - lineHeight function 347
  - of cast members 291
  - of field cast members 463
  - of lines in field cast members 347
- hiding cast member controllers 158
- highlighting text 293
- HTTP headers, date last modified 426
- HTTP proxy servers, setting values of 516
- I**
- identity transform 301
- idle time, length of 302
- if...then...else statements 305, 443
- image object
  - copying 207
  - drawing in 204
  - size of 539
- importing 314
- inker
  - colors of 151
  - modifier 34
- inks
  - ink of sprite property 316
  - trails effect 693
- inserting frames 319
- installing menus defined in field cast members 320
- integers
  - maxInteger property 370
  - random 532
- Internet
  - downloading cast members from 373
  - playing Shockwave movies from 287
  - preloading files from 203, 506
- Internet files, MIME types and 427
- interpreters for Lingo 567
- intersecting sprites 629
- items, separating 329
- K**
- keyboard, on timeOut event handler and 454
- keyframes
  - sprites and 701
  - trackNextKeyTime property 689
  - trackPreviousKeyTime property 690
- keys
  - Alt key (Windows) 456
  - assigning scripts to 337
  - Backspace key (Windows) 86
  - Command key (Macintosh) 153

- keys *continued*
  - Control key (Macintosh) 157
  - Control key (Windows) 153, 157
  - Delete key (Macintosh) 86
  - Enter key 220
  - last key pressed 330, 332, 336
  - lastEvent function 340
  - lastKey function 341
  - on keyDown event handler 679
  - on keyUp event handler 336
  - Option key (Macintosh) 456
  - RETURN character constant 559
  - Shift key 610
  - Tab key 659
- L**
- labels 255, 338
- launching applications 454
- less than operator (<) 50
- less than or equal to operator (<=) 50
- level of detail (LOD)
  - modifier 35
  - modifier properties 357
- lights 79
  - ambient light 193
  - directional light 196
  - Lingo for 35
- line spacing for cast members 348
- line wrapping 741
- linear lists
  - adding to 60, 65
  - appending to 77
  - creating 350
  - deleting values from 188
- lines
  - continuation symbol (~) 54
  - distance of 349
  - drawing in image object 204
  - height of 347
  - in cast members 347
- Lingo
  - 3D elements, by category 71
  - errors 219
  - interpreters 567
  - Xtra extensions 449
- linked
  - cast members 236
  - movies 237, 585
  - scripts 350
- list brackets ([ ]) 52
- list of sprite references 583

- list operators ([]) 52
- listing frame labels 338
- lists
  - adding to linear lists 60, 65
  - adding to property lists 65
  - appending to 77
  - copying 207
  - count function 161
  - counting items in 161
  - creating linear lists 350
  - cue point names 169
  - cue point times 170
  - deleting all items in 184
  - deleting items or values from 184, 188
  - findPos command 242
  - findPosNear command 243
  - getProp command 263
  - getAt command 264
  - getLast command 271
  - getOne command 274
  - getPos command 277
  - getProp command 278
  - getPropAt command 278
  - identifying items in 263, 264, 271, 274, 277, 278
  - ilk function 307
  - list operators ([]) 52
  - maximum value in 370
  - minimum value in 383
  - of movie windows 739
  - position of properties in property lists 242, 243
  - replacing property values from 598, 604
  - setProp command 598
  - setAt command 599
  - setProp command 604
  - sorting 616
  - types of 307, 351
  - value of parameters in 465
  - windowList property 739
- literal quotation mark (") 530
- loading cast members
  - cancelIdleLoad command 120
  - determining if loaded 351
  - displaying information when 687
  - finishIdleLoad command 243
  - idleLoadDone function 303
  - idleLoadPeriod property 303
  - idleLoadTag system property 304
  - idleReadChunkSize property 304
  - preloading cast members 503, 505

- loading cast members *continued*
  - purge priority of cast members 521
  - unloading cast members 707, 708
- local variables 523, 611
- location
  - of cast members 128
  - of mouse pointer 400, 408, 409
  - of sprites 155, 353, 354, 355, 629, 630
  - of Stage on desktop 633, 634, 635
- location number of sprites 376
- log files of Message window display 688
- logarithm functions 227, 358
- logical constants
  - FALSE 234
  - TRUE 700
- logical expressions 73
- logical negation of expressions 442
- long time format 677
- loops
  - loop keyword 359
  - next repeat keyword 439
  - repeat with keyword 553
  - repeat with...down to keyword 554
  - repeat with...in list keyword 555

## M

- Macintosh computers
  - beep sound and 87
  - Lingo interpreters 567
  - platform system property 482
  - restarting 558
  - turning off 613
- marker labels, frames associated with 338
- markers
  - before and after frames 368
  - getting list of 369
  - go loop command 285
  - going to next marker 286
  - going to previous marker 286
  - loop keyword 359
  - next keyword 438
- marking end of handlers 216
- Media Control Interface (MCI) 371
- memory
  - allocated to program 379
  - determining if cast members are loaded 351
  - free 261, 379, 413, 531
  - preloading cast members 503, 505
  - preloading movies into 506
  - ramNeeded function 531
  - required to display frames 531

- memory *continued*
  - size of free blocks 261
  - unloading cast members from 707, 708
  - unloading movies from 708
- menu item properties
  - checkMark of menuItem 132
  - enabled of menuItem 214
  - name of menuItem 420
  - script of menuItem 583
- menu items
  - checked menu items 132
  - scripts executed by 583
  - selecting 214
  - setting text in 420
- menu properties
  - name of menu 420
  - number of menus 449
- menus
  - defining custom menus 379
  - in current movie 449
  - installing menus defined in field cast members 320
  - name of menu property 420
- mesh
  - colors of 149
  - Lingo for 38
- mesh deform modifier 36
- Message window
  - displaying expression results in 522
  - displaying global variables 611
  - writing display to log files 688
- messages
  - alert command and 66
  - custom event messages for sprites 594, 595
  - error 266, 268
  - event 454
  - idle time and 302
  - invoking handlers with 116
  - passing 469
  - sending to child's ancestor 117
  - stage system property 632
  - tell command 661
- methods
  - exiting 225
  - marking end of handlers 216
  - return keyword 560
- MIAW. *See* movie in a window (MIAW)
- MIME files 288, 427
- mipmapping 527
- mod (modulus) operator 384
- model resources, Lingo for 36
- models
  - Lingo for 36
  - selecting, Lingo for 39
- modifiers 63
  - bonesPlayer 32
  - inker 34
  - keyframe player 35
  - level of detail (LOD) 35, 357
  - Lingo for applying 37
  - mesh deform 36
  - subdivision surfaces 588
  - subdivision surfaces (SDS) 41
  - toon 41
- monitors
  - centering Stage on 126
  - color depth of 148
  - size and position of 192
- motion 395
  - playback 484
  - timing 172
- mouse clicks
  - assigning scripts for 398, 407
  - clickLoc function 137
  - clickOn function 138
  - determining if mouse button is pressed 398, 404
  - doubleClick function 202
  - emulateMultiButtonMouse property 214
  - lastClick function 340
  - lastEvent function 340
  - mouseDown function 398
  - mouseUp function 404
  - on mouseDown event handler 397, 681
  - on mouseEnter event handler 402
  - on mouseLeave event handler 402
  - on mouseUp event handler 408
  - on mouseUpOutside event handler 408
  - on mouseWithin event handler 408
  - on rightMouseUp event handler 564
  - right mouse button status 564, 565
  - stillDown function 644
- mouse pointer position 400, 401, 403, 408, 409
- movie in a window (MIAW) 237, 254, 661, 739
  - hiding 78
  - minimizing 78
- movie names 412, 415, 472
- movie properties
  - allow zooming 71
  - center of member 125
  - controller of member 158
  - idleHandlerPeriod 302

- movie properties *continued*
  - idleReadChunkSize 304
  - paletteMapping 463
  - scoreSelection 582
  - scriptsEnabled of member 585
  - updateLock 710
  - updateMovieEnabled 710
  - videoForWindowsPresent 722
- movie windows
  - drawRect of window property 205
  - frontmost movie in 262
  - list of 739
  - running objects as movies in 739
- movies
  - color depth settings (Macintosh) 657
  - deleting frames from 186
  - determining version created with 414
  - file size of 414
  - finding 274
  - go loop command 285
  - going to frames 284
  - idle time and 301
  - inserting frames 319
  - last frame in 341
  - length of in time 416
  - movie names 412, 415, 472
  - number of casts in 444
  - number of menus in 449
  - on stopMovie event handler 652
  - pausing 157, 473, 477
  - playback rates 416
  - playing from a marker 286
  - playing with play command 483, 487
  - preloading 506
  - properties, Lingo for 37
  - responding to events 385
  - run mode of 572
  - saving 577, 710
  - Score associated with 581
  - Shockwave 229, 230, 287
  - stopping movie playback 183, 290
  - unloading from memory 708
  - unused space in 413
  - Xtra extensions available to 449
- moving
  - cast members 410
  - sprites 410
  - windows 411
- multiplication operator (\*) 47

## N

- names
  - of cast members 419
  - of casts 419
  - of movies 412, 415, 472
- naming windows 421
- natural logarithm functions 227, 358
- network operations
  - canceling 424
  - errors during 425
  - MIME types and 427
  - text returned by 428
- network servers, retrieving text from files on 272
- new line symbol (↵) 54
- newsprint shader, Lingo for 40
- nodes
  - deleting 185
  - managing, with Lingo 37
  - parent/child, Lingo for 33
- normals 263, 381, 440
  - list of 441
- not equal operator (<>) 50
- not logical operator 442
- number sign (#) 43, 525
- numbers
  - cast member number of frame scripts 258
  - cast numbers 444
  - converting expressions to floating-point numbers 248
  - displaying number of current frame 254
  - exponents 501
  - floating-point numbers 248, 249
  - largest supported by system 370
  - random number generation 533
  - script number assigned to sprites 584

## O

- OBJECT tags 229, 230
- objects
  - creating and removing with Lingo 33
  - deleting 189
  - removing 548
- opening
  - applications 454
  - MIME files 288
  - Shockwave movies 288
  - windows 455
- operators
  - mod 384
  - not 442
  - or 457
  - sprite...intersects 629

- Option key (Macintosh) 456
- Option-Return character (↵) 54
- or logical operator 457
- overlays
  - inserting 319
  - manipulating with Lingo 32
- P**
- painter shader, Lingo for 40
- palette channels as puppets 517
- palettes
  - assigning to cast members 463, 464
  - in current frame 256
  - remapping 463
  - Score color assigned to sprites 582
- parameters
  - counting parameters sent to handlers 466
  - in lists 465
- parent nodes, Lingo for 33
- parent scripts
  - ancestor property 72
  - me keyword 372
  - objects created by 452
- parentheses operator () 47
- parents 466
- particle systems
  - distribution 200
  - gravity 284
  - Lingo for 38
- particles, Lingo for 38
- Paste command, enabling 212
- pasting Clipboard contents into cast members 470
- pathname operator (@) 54
- pathnames 54, 415, 472, 589
- paths
  - applicationPath property 77
  - browsers and 110
  - searched by Director 589
- patterns, filling shape cast members with 240
- pausing movies 157, 473, 477
- pausing playback of cast members 476
- picking, Lingo for 39
- planes, Lingo for 39
- playback capabilities, determining with Lingo 37
- playback rates 416
- playing
  - AIFF files 624
  - AVI files (Windows) 624
  - digital video tracks 606
  - Shockwave movies from Internet 287
  - sound 618, 624
  - WAVE files 624

- playing movies
  - after a pause 157
  - going to a frame 284
  - going to a marker 286
  - setting frame rate 256
  - with play command 483, 487
- playing time of sound sprites 171
- plus sign (+) 48
- points
  - coordinates of 491
  - identifying 351
  - point function 491
  - positioning and sizing 366
  - type of 307
- position
  - of cast members 128
  - of mouse pointer 400, 408, 409
  - of sprites 155, 353, 354, 355, 629, 630
  - of Stage on desktop 633, 634, 635
- positioning rectangles and points 366
- pound sign (#) 43, 525
- preloading
  - cast members 303
  - files from Internet 203, 506
  - Shockwave Audio cast members 507
- primitives, Lingo for 38
- printing
  - frames 513
  - Stage 513
- projectors
  - exiting 226
  - hiding 78
  - minimizing 78
- property lists
  - adding to 65
  - deleting values from 188
  - findPos command 242
  - findPosNear command 243
  - position of properties in 242, 243
  - replacing property values from 598, 604
  - setProp command 598
  - setProp command 604
- property variables 515, 523
- proxy servers, setting values of 516
- puppets
  - palette channels as 517
  - puppet sprite blend values 95
  - sound channels as 518
  - sprites as 516, 519
  - tempo channels as 519
- purge priority of cast members 521

## Q

QuickTime 3 masks 369  
quitting  
    Director 530, 613  
    handlers 56, 225, 290  
    projectors 226  
quotation mark (") 54, 530

## R

radio buttons, checkBoxAccess property 131  
random integers 532  
random number generation 533  
rectangles  
    changing dimensions of 316  
    coordinates of 540, 625  
    defining 538  
    inflate rect command 316  
    inside function 320  
    intersect function 323  
    offsetting 453  
    positioning and sizing 366  
    type of 307  
    union rect function 707  
rects, identifying 351  
redrawing Stage 711  
reflectivity 542  
refreshing web pages 115  
registering 543  
registration points 547  
    of bitmap cast members 547  
    of sprites 354, 355  
    updating 481  
relative paths, pathname operator (@) 54  
remainders of division 384  
remapping palettes 463  
rendering 501, 550  
    determining renderer 58  
repeat loops  
    exit repeat keyword 227  
    next repeat keyword 439  
    repeat with keyword 553  
    repeat with...down to keyword 554  
    repeat with...in list keyword 555  
resolution 556, 557  
resource files  
    displaying resources in 612  
    opening 454, 456  
retrieving files 274  
retrieving text from files on network servers 272  
rollOver tests 340, 566  
rotation 571  
rounding floating-point numbers 249

## S

sampling  
    sampleRate of member property 575  
    trackNextSampleTime property 689  
    trackPreviousSampleTime property 690  
saving  
    changes to casts 576  
    movies 577, 710  
Score  
    associated with current movie 581  
    channels selected in 582  
    recording 87, 218  
    Score color assigned to sprites 582  
    updating 87, 710  
screens  
    centering Stage on 126  
    identifying mouse clicks on 137  
    size and position of 192  
scripting Xtra extensions 745  
scripts  
    assigned to cast members 585  
    assigning for keys 334, 337  
    assigning for mouse clicks 398, 407  
    attached to sprites 605  
    cast member number of frame scripts 258  
    comment delimiter (--) 45  
    debugging 611  
    in linked movies 585  
    invoking handlers in 116  
    linked 350  
    menu item execution of 583  
    objects created by parent scripts 452  
    rollOver tests in frame scripts 566  
    script number assigned to sprites 584  
    types of 586  
scrolling field cast members 586, 587  
searching for filenames 274, 589  
sending strings to browsers 228  
separating items 329  
servers, proxy 516  
shaders, Lingo for 39, 608  
shapes  
    borders of 349  
    patterns for 240, 472  
    types of 609  
shininess 613  
Shockwave Audio cast members  
    errors and 266, 268  
    pausing playback of 476  
    percentPlayed of member property 477  
    percentStreamed of member property 478  
    play member function 489

- Shockwave Audio cast members *continued*
  - preLoadBuffer member command 504
  - preloading 507
  - specifying URL for 711
  - state of 640
  - stopping playback of 647
- Shockwave movies
  - names of external parameters 229
  - number of external parameters 229
  - opening 288
  - playing from Internet 287
  - values from external parameters 230
- short time format 677
- size
  - chunkSize of member property 133
  - fixStageSize property 245
  - idleReadChunkSize property 304
  - lineSize of member property 349
  - of cast members 133, 614
  - of free blocks of memory 261
  - of monitors 192
  - of movies 414
  - of Stage 245
  - size of member property 614
- sizing rectangles and points 366
- slash sign (/) 49
- smoothness 615
- sorting lists 616
- sound
  - fading in 620
  - fading out 621
  - levels 622, 735
  - playing 618, 624
  - stopping playback 618, 624
  - turning on or off 619
  - volume control 622, 735, 736
- sound cast member properties, bitRate of member 93
- sound channels
  - as puppets 518
  - cast member 376
  - cast member assigned to first channel 259
  - cast member assigned to second channel 259
  - closing 618
  - getting status of 325, 643
  - looping 361, 362, 363
  - number of samples in 574
  - playing sound in 617
  - playlist 602
  - rewinding 561
  - stopping 646
- sound properties
  - channelCount of member 127
  - multiSound 418
  - sampleRate of member 575
  - soundEnabled property 619
  - soundLevel property (Macintosh) 622
  - volume of sound 735
  - volume of sprite 736
- sound sprites, current playing time of 171
- space character 626
- specular (3D property)
  - light property 626
  - shader property 627
- specularity 626
- spheres, Lingo for 39
- sprite...intersects operator 629
- sprite...within comparison operator 630
- sprites
  - 3D, Lingo for 40
  - active sprites 138
  - assigning cast members to 378
  - beepOn property 87
  - behavior scripts attached to 583
  - channel number of 630
  - clearFrame command 135
  - clickOn function 138
  - constrainV function 156
  - coordinates of 105, 154, 156, 342, 562, 685
  - counting tracks on digital video sprites 688
  - cue points and 327, 395
  - current playing time of 171
  - currentSpriteNum property 171
  - cursor resource used when pointer is over 175
  - custom event messages for 594, 595
  - deleting frame's sprite contents 135
  - displaying digital video cast members in 167
  - editing field sprites 214
  - keyframes and 701
  - media in digital video sprite tracks 693
  - mouseChar function 396
  - mouseMember function 404
  - moving 410
  - moving on Stage 410
  - on cuePassed event handler 219
  - on endSprite event handler 219
  - playing tracks on 689
  - position of 155, 353, 354, 355, 629, 630
  - puppet sprite blend values 95
  - puppets and 516, 519
  - registration points of 354, 355



sprites *continued*

- rollOver tests 566
- script number assigned to 584
- scripts attached to 605
- specifying location number 376
- sprite keyword 629
- starting time of movies in sprite channels 691
- stop time of tracks 691
- stretching 653
- trails effect 693
- visibility of 726

square brackets ([]) 52

Stage

- centering on monitor 126
- fixStageSize property 245
- position of on desktop 633, 634, 635
- printing 513
- redrawing 711
- setting color of 633
- size of after loading movies 245
- sprite position on 353
- updating 711

starting

- applications 454
- character in selections 594
- Score update sessions 87

starts comparison operator 637

statements, if...then...else 305

stopping

- event messages 454
- movie playback 183, 290
- playback of cast members 647
- sound playback 618, 624

stream status reporting 662

streaming, Lingo for 40, 479

stretching sprites 653

string constant (") 54

strings

- ASCII codes 130, 450
- char...of keyword 127
- chars function 129
- comparing 156
- converting expressions to 653
- counting characters in 448
- counting items in 445
- counting lines in 446
- counting words in chunk expressions 448
- date last modified 426
- deleting chunk expressions 184
- displaying in browser window 428

strings *continued*

- do command 200
  - EMPTY character constant 214
  - ending character selections 593
  - expressions as 653
  - field keyword 235
  - highlighting 293
  - in field cast members 664
  - integer function 321
  - item...of keyword 328
  - last function 339
  - length function 343
  - line...of keyword 346
  - numerical value of 717
  - offset function 452
  - put...after command 522
  - put...before command 523
  - put...into command 523
  - quotation marks and 54
  - selecting 592
  - sending to browsers 228
  - starting character in selections 594
  - starts comparison operator 637
  - writing to files 603
- subdivision surfaces (SDS)
- modifier 41, 588
  - properties 588
- subtraction operator (-) 44
- symbol definition operator (#) 43, 525
- symbols
- expressions and 658
  - strings and 657
  - symbol definition operator (#) 43, 525
- system beep elements
- alert command 66
  - beep command 87
- system clock, time function 677
- system properties
- activeCastLib 59
  - activeWindow 59
  - checkBoxType 131
  - deskTopRectList 192
  - digitalVideoTimeScale 194
  - emulateMultiButtonMouse 214
  - floatPrecision 249
  - frontWindow 262
  - idleLoadMode 303
  - idleLoadTag 304
  - keyPressed 336
  - multiSound 418

- system properties *continued*
  - platform 482
  - rightMouseDown 564
  - rightMouseUp 565
  - stage 632
- system properties, Lingo for 37

## T

- TAB character constant 659
- Tab key 659
- tabbing order, autoTab of member property 83
- tempo
  - assigned to frames 260
  - settings 260
  - tempo channels as puppets 519
- text
  - ASCII codes 130, 450
  - char...of keyword 127
  - charPosToLoc function 128
  - chars function 129
  - comparing strings 156
  - concatenation operators (& or &&) 45, 46
  - converting expressions to strings 653
  - counting items in 445
  - counting lines in 446
  - counting number of characters in 448
  - counting words in chunk expressions 448
  - deleting chunk expressions 184
  - do command 200
  - EMPTY character constant 214
  - ending character in selections 593
  - expressions as strings 653
  - field keyword 235
  - highlighting 293
  - item...of keyword 328
  - last function 339
  - length function 343
  - line...of keyword 346
  - numerical value of strings 717
  - offset function 452
  - put...after command 522
  - put...before command 523
  - put...into command 523
  - retrieving from files on network servers 272
  - returned by network operations 428
  - selecting 592
  - selecting words in 740
  - starting character in selections 594
  - starts comparison operator 637
  - strings in field cast members 664
- text boxes for cast members 108

- textures 665
  - coordinates 381
  - Lingo for 41
- ticks
  - converting time to 676
  - lastClick function 340
  - lastKey function 341
  - lastRoll function 341
  - movieTime of sprite property 416
  - number of before timeout occurs 680
  - ticks function 676
  - timeoutLength property 680
  - timer property 682
- time
  - converting frames to duration in time 259
  - converting to ticks 676
  - time formats 677
  - time function 677
  - units of measurement 194, 683
- timeout object
  - determining name 421
  - returning 678
  - sending events to child object 660
- timeouts, Lingo for 681
- toon
  - colors of 151
  - modifier 41
- tracks, playing 606
- transforms
  - angles 83
  - inverting or reversing 324
  - Lingo for 42
- transition cast members
  - assigned to current frame 260
  - duration of 208
  - properties, chunkSize of member 133
- transitions
  - between frames 260, 520
  - puppetTransition command 520
  - transitionType of member property 696
  - types of 696
- translations 696
- TRUE logical constant 700
- turning digital video cast member play on or off 721, 726
- turning sound on or off 619
- tween mode 701
- typeface 250, 251

## U

- units of measurement for time 194, 683
- unloading
  - cast members from memory 707
  - movies from memory 708
- updating
  - frames 709
  - registration points 481
  - Score 87, 218, 710
  - Stage 711
- URLs, Shockwave Audio cast members and 711
- user data 715

## V

- values, comparing 121
- variables
  - global variables 136, 282, 611
  - local variables 611
  - property variables 515
  - voidP property 735
- vectors, Lingo for 42, 718
- video
  - deleting frames from movies 186
  - pausing movies 157
  - preloading movies 507
  - stopping movie playback 183, 290
- Video for Windows software 722
- VOID constant 734

## W

- WAVE files 624
- web pages, refreshing 115
- width of cast members 737
- window properties
  - activeWindow 59
  - drawRect of window 205
  - fileName of window 237
  - modal of window 385
  - name of window 421
  - sourceRect 625
  - title of window 683
  - titleVisible of window 683
  - visible of window 726
  - windowList 739
  - windowType of window 740
- windows
  - active window 59
  - closing 141, 254
  - coordinates of 541
  - displaying strings in browser windows 428
  - events outside of 385

## windows *continued*

- forget window command 254
- frontmost movie in 262
- moving windows behind other windows 411
- moving windows in front of other windows 411
- naming 421
- on activateWindow event handler 59
- on closeWindow event handler 219
- on deactivateWindow event handler 219
- on moveWindow event handler 455
- on openWindow event handler 455
- on zoomWindow event handler 454
- opening 455
- stage property 632
- visibility of 726
- window keyword 738

Windows computers

- beep sound and 87
- platform system property 482
- turning off 613

words in chunk expressions 448

world units 462

wrapping lines 741

## X

- XCMDs and XFCNs (Macintosh) 456
- XCOD resources 456, 612
- Xlibrary files
  - closing 142
  - displaying Xtra extensions and XObjects in 612
  - opening 456
- XObjects
  - and numerical value of strings 717
  - displaying in Xlibrary files 612
  - opening 456
- Xtra extensions
  - available to movie 449
  - creating 429
  - displaying in Xlibrary files 612
  - name of xtra property 421
  - numerical value of strings and 717
  - objects created by 452
  - xtra function 745

## Z

- zooming
  - allowing 71
  - effects 749

